Indiana University
Computer Science Department

Technical Report 600
http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR600

Scheme 2004

Proceedings of the Fifth Workshop on Scheme and Functional Programming

September 22, 2004
Snowbird, Utah

Olin Shivers and Oscar Waddell, editors

Sponsored by the Association for Computing Machinery's Special Interest Group on Programming Languages (ACM/SIGPLAN)

Preface

This report contains the papers presented at the Fifth Workshop on Scheme and Functional Programming, on September 22, 2004, in Snowbird, Utah.

The purpose of the workshop is to discuss experience with, and future developments of, the Scheme programming language, as well as general aspects of Computer Science loosely centered on the general theme of Scheme. The intention of the steering committee is that the workshop provide an annual focal point where the Scheme community can gather and share ideas: researchers, educators, implementors, programmers, hobbyists, and enthusiasts of all stripes—all welcome.

Eleven papers were submitted in response to the workshop's call for papers. Paper submission and review was conducted via electronic mail. Each paper was read by at least three reviewers including at least two members of the program committee. We are grateful to Olivier Danvy, Kent Dybvig, Martin Gasbichler, Eric Knauel, and Bradley Lucier for their service as outside reviewers.

Several others helped with the planning for the workshop. Mayer Goldberg lent his type-setting expertise to the production of this technical report. Matthew Flatt of the University of Utah allowed himself to be drafted for local arrangements without demur or complaint. Chris Okasaki, ICFP general chairman, and Franklyn Turbak, ICFP workshop chairman, were consistently helpful throughout the process. The Scheme workshop steering committee provided advice and general counsel during the planning of the workshop. We are thankful for all of this support and assistance.

Olin Shivers and Oscar Waddell,
For the program committee

Contents

Scheme Program Documentation Tools

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.aau.dk

Abstract

This paper describes and discusses two different Scheme documentation tools. The first is SchemeDoc, which is intended for documentation of the interfaces of Scheme libraries (APIs). The second is the Scheme Elucidator, which is for internal documentation of Scheme programs. Although the tools are separate and intended for different documentation purposes they are related to each other in several ways. Both tools are based on XML languages for tool setup and for documentation authoring. In addition, both tools rely on the LAML framework which—in a systematic way—makes an XML language available as a set of functions in Scheme. Finally, the Scheme Elucidator is able to integrate SchemeDoc resources as part of an internal documentation resource.

1 Introduction

Program documentation tools are important for all kinds of non-trivial programming tasks. In a general sense, program documentation tools make it possible to produce important information for programmers who apply a program library, and for future developers of a program. In this paper we are concerned with program documentation for Scheme developers. End user documentation is not an issue in this paper.

We discuss two documentation tools for Scheme. The first, SchemeDoc, is a tool for documenting library interfaces—also known as application programmers interfaces (APIs). The documentation produced by SchemeDoc is intended for Scheme programmers who apply the documented Scheme library. The second, the Scheme Elucidator, is a tool for documentation of the internal details of a Scheme program. The documentation produced by the Scheme Elucidator—called an elucidative program—is typically intended for future maintainers of the program. Elucidative Scheme programs may, however, be targeted towards any reader with an interest in understanding the program. As such, the Scheme Elucidator can be used whenever there is a need to *write about* a Scheme program, for educational, tutorial, or scientific purposes.

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

SchemeDoc and the Scheme Elucidator share a number of properties. The input formats of both tools are defined as XML languages, with use of XML DTDs (Document Type Definitions) [1]. In simple cases, the input formats are relatively small setup files that hold a number of processing attributes, whereas in other cases, substantial amounts of documentation is authored within the XML documents. Both tools are part of LAML (see Section 2) and as such the full expressiveness of Scheme can be used in the XML-related parts of both SchemeDoc documentation and in elucidative programs. Finally, both tools generate web output, in terms of XHTML files.

In this paper we give overall and concise descriptions of the tools. More detailed descriptions can be found on the web [16, 20]. As part of the discussions we point out relevant details in the implementation of the tools. In addition we compare the tools with other similar documentation tools for Scheme.

The paper is structured as follows. In Section 2 we summarize the most basic properties of LAML, which is the common underlying platform of both tools. In Section 3 we discuss SchemeDoc. This includes a discussion of similar tools for documentation of Scheme libraries. In Section 4 we discuss the Scheme Elucidator. The main contributions and the conclusions are summarized in Section 5. It is possible to skip Section 3 in case the reader is only interested in documentation of internal programs aspects with the Scheme Elucidator. The programs and documentation that are discussed in this paper are all available as web resources [21].

2 LAML Background

Both tools described in this paper rely on LAML (Lisp Abstracted Markup Language), and we will therefore in this section provide a brief summary of LAML. For more information about LAML please consult the paper *Web Programming in Scheme with LAML* [17] and the LAML home page [18].

From an overall perspective, LAML attempts to come up with natural Scheme-based counterparts to the most important aspects of XML. The main purpose of LAML is to make XML languages available as sets of Scheme functions. With this, an XML document becomes a Scheme expression. As a consequence, the power of Scheme is available anywhere in a document, and at any time during the authoring process. We refer to this situation as *programmatic authoring* [15].

The set of Scheme functions that corresponds to the elements of an XML language *L* is called a *mirror of L in Scheme*. Each element of an XML language is represented as a Scheme function. When applied, these functions generate an internal format (ASTs repre-

sented as lists) and they carry out a comprehensive documentation validation at run time (document processing time).

LAML provides an XML DTD parser and a mirror generation tool. These tools have been used to generate validating mirrors of XHTML, SVG and a number of more specialized XML languages (such as the SchemeDoc language and the Elucidator language discussed in this paper).

Web authoring with LAML is supported by a set of convenient Emacs editor commands. No specialized lexical Scheme conventions are used. As an example, the sample XML fragment

```
<book id = "sicp">
  <title>Structure and Interpretation
         of Computer Programs</title>
  <authors>
    <author>Abelson</author>
    <author>Sussman</author>
  </authors>
</book>
```

can be written as the Scheme expression

```
(book 'id "sicp"
  (title "Structure and Interpretation
          of Computer Programs")
  (authors (author "Abelson") (author "Sussman"))
)
```

provided that the mirror of the book description language is loaded on beforehand.

The parameters to each mirror function are interpreted relative to the *LAML parameter passing rules* [17], which can be summarized as follows: An attribute is a symbol; an attribute value is the string following a symbol; other strings represent textual element content items; lists are recursively unfolded. If relevant, white space is always provided in between element content items unless explicitly suppressed by a distinguished value (`#f` usually bound to the variable named `_`). As a consequence of the list unfolding rule, the expression

```
(authors (map author (list "Abelson" "Sussman")))
```

is equivalent to

```
(authors (author "Abelson") (author "Sussman"))
```

The definition of XML languages, and their mirrors in Scheme, can be seen as a *linguistic abstraction process*. With use of the higher-order function `xml-in-laml-abstraction` it is, in addition, possible for the author to define functions that use LAML parameter passing rules. Seen in contrast to the linguistic abstractions, such functions are called *ad hoc abstractions*.

LAML works on a variety of different Scheme Systems on Unix and Windows. Therefore the documentation tools discussed in this paper can be used together with many different Scheme systems on both platforms.

3 SchemeDoc

As stated in the introduction, SchemeDoc is a tool for creation of web documentation of programmatic interfaces of Scheme programs, most notable the interfaces of program libraries. Many programmers are familiar with web documentation of programmatic interfaces from the success of Javadoc [2, 29]. As Javadoc, SchemeDoc supports extraction of documentation from distinguished *documentation comments* in source programs. In addition, SchemeDoc allows manual authoring of the documentation, and documentation of XML mirror functions in Scheme. In the section 3.1 below we describe these possibilities.

3.1 SchemeDoc operational modes

SchemeDoc can be used in four operational modes:

- **Source Extraction mode.**
  The documentation is extracted from distinguished documentation comments in a Scheme source program.
- **Manual mode.**
  The documentation is authored manually, in an XML format with use of LAML.
- **XML DTD mode.**
  The documentation is extracted from a parsed XML DTD, typically with the purpose of documenting the mirror of the XML language in Scheme.
- **Augmented XML DTD mode.**
  A mixture of the XML DTD mode and the manual mode. Documentation, which is not present in the DTD is authored manually and merged with the extracted DTD documentation.

The Source Extraction mode relies on the concepts of comment blocks and documentation comments. A *comment block* is a sequence of consecutive Scheme comment lines (each of which is initiated with a semicolon). A *documentation comment* is a comment block which, by means of a given commenting style, is set apart from "ordinary comments".

Documentation comments are classified as either definition comments, documentation sections, or documentation abstracts. A *definition comment* precedes and documents a Scheme definition. A *documentation section* describes common properties of the set of definitions that follows the section comment. A *documentation abstract* gives an initial and overall description of a Scheme source file.

SchemeDoc supports two different *commenting styles* for identification of documentation comments: multi-semicolon style and documentation-mark style. Using *multi-semicolon style*, each documentation comment line is initiated with two, three or four semicolons, supporting definition comments, documentation sections, and documentation abstracts respectively. Using *documentation-mark style*, a documentation comment is identified with occurrences of a distinguished character (per default '!') at the start of the first comment line in a comment block. Definition comments use a single mark, documentation sections use two marks, and documentation abstracts use three exclamation marks. Until recently, all LAML software has been documented using multi-semicolon style.

Within documentation comments, a *little markup language* is used to provide additional structure. SchemeDoc uses dot-initiated documentation keywords together with a line-oriented organization. These elements of SchemeDoc are, to a large degree, modelled directly after similar systems, such as Javadoc [2, 29] and Doxygen [30].

As a concrete illustration of SchemeDoc in Source Extraction mode with use of multi-semicolon documentation comments, the Scheme

```
;;;; .title SchemeDoc Demo
;;;; .author Kurt Normark
;;;; .affiliation Aalborg University, Denmark
;;;; This is a brief example of a Scheme
;;;; program with multi-semicolon SchemeDoc comments.

; This comment is not extracted.

;;; Factorials.
;;; .section-id fac-stuff
;;; This section demonstrates a plain function.

;; The factorial function. Also known as n!
;; .parameter n An integer
;; .pre-condition n >= 0
;; .returns n * (n-1) * ... * 1
(define (fac n)
 (if (= n 0) 1 (* n (fac (- n 1)))))

;;; List selection functions.
;;; .section-id list-stuff
;;; This section demonstrates two aliased functions.

;; An alias of car.
;; .returns The first component of a cons cell
;; .form (head pair)
;; .parameter pair A cons cell
(define head car)

;; An alias of cdr.
;; .returns The second component of a cons cell
;; .form (tail pair)
;; .parameter pair A cons cell
(define tail cdr)
```

Figure 1: *A Scheme program with documentation comments in multi-semicolon style.*

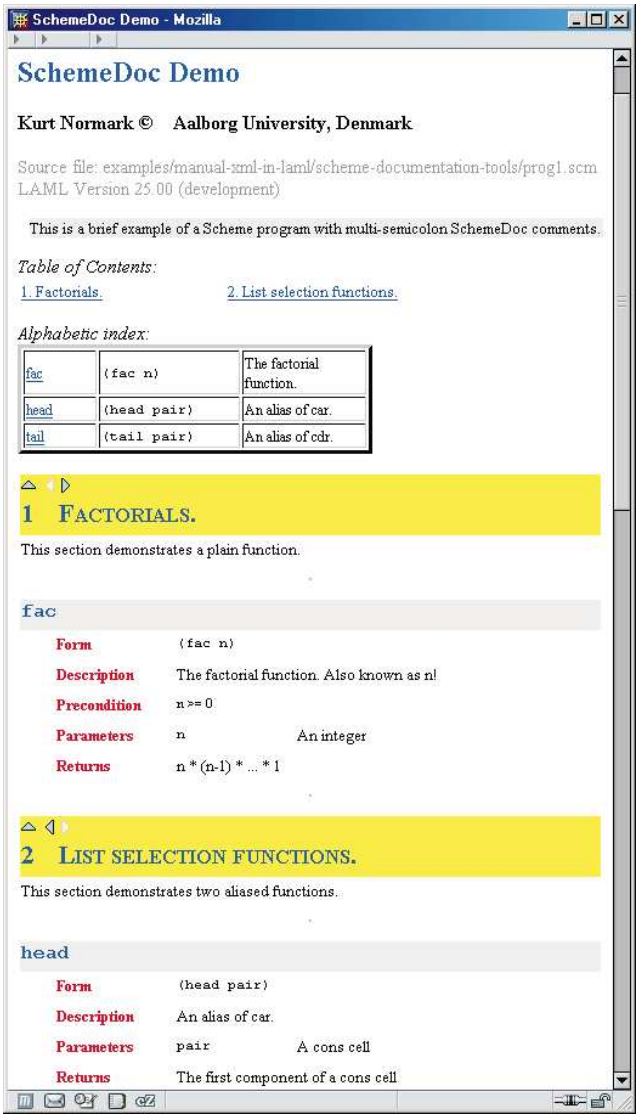


Figure 2: *A partial presentation of the SchemeDoc documentation from Figure 1.*

Program in Figure 1 gives rise to the extracted documentation, shown partially in Figure 2. (The same example is shown with use of documentation-mark style at the web resource page [21] of this paper). The figure illustrates a single documentation abstract, two documentation sections, and three definition comments. SchemeDoc ignores one-semicolon comments. In Figure 1 we illustrate the `title`, `author`, and `affiliation` tags in the documentation abstract. In the section comments, we illustrate the `section-id` tag, which is used for generation of an anchor name in HTML. In the definition comments, we illustrate the `form`, `parameter`, `pre-condition`, and `returns` tags. The `form` tag is used in situations where the actual calling form does not appear as a constituent of the definition.

SchemeDoc can deal with nested documentation comments. More specifically, definition comments and documentation sections are extracted from the definitions, which are documented by means of definition comments. In the current version of SchemeDoc, we only handle two levels of nested documentation comments.

XML DTD mode can, in general, be used for documentation of an XML DTD, which has been parsed with the LAML DTD parser [18]. The documentation of an XML DTD presents the DTD after full expansion of the parameter entities [1] (textual macros in the DTD). Use of parameter entities is convenient in order to reduce the complexity of the DTD authoring process, but they make it difficult to read the DTD. Activation of SchemeDoc on the parsed XML DTD file leads to a straightforward presentation of the XML elements, primarily in terms of the XML content model and information about the attributes. The presentation of content models provides for easy navigation to constituent elements, and to elements in which the current element appears as a constituent. The XML DTD mode of SchemeDoc is of particular importance for documentation of the major and well-known XML languages, such as the different versions of XHTML and SVG, for which LAML provides mirrors in Scheme.

The augmented XML DTD mode makes it possible to combine manually authored contributions with the documentation extracted from the XML DTD. In that respect, this mode is a mixture of the Manual mode and the XML DTD mode, as described above. More specifically, SchemeDoc is able generate an initial documentation file (in the format used in Manual mode). By filling in the con-

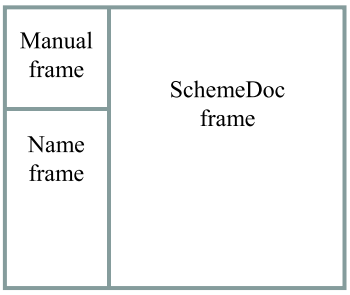


Figure 3: *The frame layout of a SchemeDoc index.*

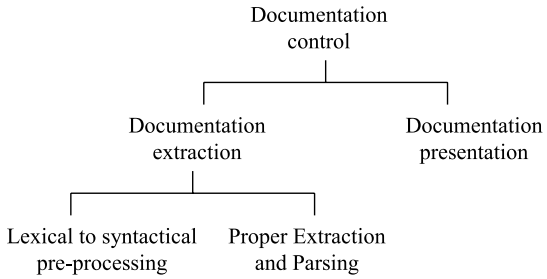


Figure 4: *The decomposition of the SchemeDoc tool.*

tents of element and attribute descriptions, the intuitive meaning of the elements can be documented. It is, in addition, often helpful to add some sectioning to provide for better structure and overview. At SchemeDoc processing time, the manually authored documentation is added to the information from the parsed XML DTD. In this way, the information from the parsed XML DTD always controls the final documentation. All substantial LAML document styles, including SchemeDoc and the Scheme Elucidator (see Section 4) are documented by use of SchemeDoc in Augmented XML DTD mode. Examples of such documentation can be seen via the LAML home page [18].

3.2 SchemeDoc Indexing

A collection of SchemeDoc manuals can be indexed and organized with use of the SchemeDoc Indexing tool. Based on an enumeration of a number of SchemeDoc manuals, this tool produces a browser with three frames (see Figure 3). The browser is made available as a frameset in XHTML. The Manual frame lists the involved SchemeDoc manuals. The Name frame shows a sorted list of the defined names from a selected manual, or from all the manuals taken together. The SchemeDoc frame shows selected details from the selected manual.

The SchemeDoc indexing tool is also able to produce a useful index of the Scheme Report [6] (either R4RS or R5RS). The list of Scheme procedures and syntactic forms can either be shown separately, or it can be merged with the names from the involved SchemeDoc manuals.

3.3 Tool Support

The SchemeDoc tool can be used in several different ways. The primary way is to execute a LAML script, which parameterizes SchemeDoc appropriately. The LAML script, which extracts and creates the documentation in Figure 2 from the Scheme source program in Figure 1, is shown here:

```
(load (string-append laml-dir "laml.scm"))
(laml-style "xml-in-laml/manual/manual")

(manual
 (manual-front-matters
  'css-prestylesheet "compact"
  'css-stylesheet "original"
  'laml-resource "true"
  'documentation-commenting-style "multi-semicolon"
 )

 (manual-from-scheme-file 'src "../prog1.scm")
)
```

The LAML script can be executed via the operating system shell, from a Scheme read-eval-print loop, or via an Emacs command. Following the loading of `laml.scm` (first line) and the SchemeDoc manual stuff (second line) the `manual` clause contains tool setup parameters (the `manual-front-matters` clause) and specification of tool functionality (the `manual-from-scheme-file` clause). In this particular example, the element `manual-from-scheme-file` causes SchemeDoc to be used in Source Extraction mode. Typically, we organize LAML scripts, like the one shown above, in a `man` subdirectory of the Scheme source file directory.

SchemeDoc can also be used without LAML scripts. As one alternative, it is possible to activate a `schemedoc` procedure from a (LAML-enabled) Scheme read-eval-print loop. Another alternative is to activate SchemeDoc on a Scheme source file by use of the `schemedoc` command from Emacs. This can be done by `M-x schemedoc`, or via the menu attached to Scheme mode in Emacs. In these cases, the `manual-front-matters` attributes can be given in the documentation abstract comment. In that way, the SchemeDoc setup parameters (processing options) can be given as part of the Scheme source program.

3.4 Implementation Issues

The Source Extraction mode of SchemeDoc is implemented as documentation extraction followed by documentation presentation, both of which are managed by a documentation control layer. This architecture is illustrated in Figure 4. The top-level control part, the documentation extraction part, and the documentation presentation part are physically separated in the LAML software package. All parts are written in Scheme.

The documentation control layer manages the LAML authoring format (the mirror of the XML SchemeDoc language in Scheme). As it appears in Figure 4, the documentation extraction layer is subdivided in a source file pre-processing part and a proper extraction part. In the source file pre-processing part, lexical comments are transformed to syntactic comments. As an example, the lexical comment `;This is a comment` is transformed to a list like `(comment 1 "This is a comment")`. The second element of the list represents the categorization of the comment (here 1-semicolon comment).

With this pre-processing it is a matter of simple Lisp parsing (reading) to access the documentation comments in other parts of SchemeDoc. The proper extraction and parsing part examines the comment forms and parses the comment strings relative to the documentation markup language. The documentation extraction phase delivers an internal representation in terms of a list of association

lists. As an example, the contribution from the `fac` function in Figure 1 is the (slightly elided) association list

```
((kind "manual-page")
 (parameters (parameter "n" "An integer."))
 (description "The factorial function...")
 (pre-condition "n >= 0.")
 (returns "n * (n-1) * ... * 1")
 (title "fac")
 (form (fac n))
)
```

The documentation presentation part generates an XHTML document (with use of CSS styling) from the information in the list of association lists. The internal manual representation is written to an auxiliary file with extension '`manlsp`' such that other tools easily can access the details of a SchemeDoc manual. This information is essential for the SchemeDoc indexing tool (see Section 3.2). The Scheme Elucidator (see Section 4) does also make use of the internal manual representation.

3.5 Similar work

There exists a number of tools which are similar to SchemeDoc. Schematics SchemeDoc [26] is work in progress, primarily oriented towards PLT Scheme, and only scarcely documented. As a novel aspect, this tool uses Scheme lists for markup purposes within documentation comments. Documentation comments are initiated with an exclamation mark. The following slightly elided example (from the web site of Schematics SchemeDoc) illustrates this:

```
;;!
;; (function map
;;   (form (map fn list) -> list)
;;   (contract  ... -> ...)
;;   (example (map (lambda (elt) ...) ...))
;;
;; Apply fn to every element of list.
(define (map fn list) ...)
```

Scmdoc [27], which is a contribution to Bigloo, uses documentation comments distinguished by an exclamation mark after the semicolons of each comment line. Scmdoc is documented clearly and concisely. Directives within a Scmdoc documentation comment are prefixed with '`@`'. The following example is from the Scmdoc documentation:

```
;! @description
;! The documentation generation function.
;! @param iport The input port.
;! @param oport The output port.
;! @return Returns <CODE>#f</CODE>.
(define (scm->html iport oport) ...)
```

Docscm [3] is another similar system, which generates DocBook XML. Docscm is implemented in the Chicken Scheme system. Here is an example, which illustrates that '`@`' is used to distinguish documentation comments from other comments.

```
;;@
;; Returns <varname> arg <varname> * 2
(define (double arg) (* arg 2))
```

In addition, Docscm supports a number of directives prefixed with '`@`', and it supports a notion of documentation sections.

It should be noticed that the documentation-mark style in source extraction mode of LAML SchemeDoc is similar to the commenting conventions supported by Schematics SchemeDoc and Docscm.

Finally, Schmooz [4] is a Texinfo markup language embedded in Scheme comments. Schmooz works with Jaffer's SCM, and it is used to extract documentation from Scheme source files for subsequent Texinfo processing. Schmooz has been used for documentation of SLIB [5].

4 The Scheme Elucidator

The Scheme Elucidator can be used to *write about* a program. Documentation generated by the Scheme Elucidator typically addresses the internal program details, as a contrast to SchemeDoc documentation of the external interface. Elucidative programs are related to literate programs [9], at least in the sense that both can be considered as *program essays*. Whereas a literate program organizes program fragments as constituents of the documentation, programs and documentation are represented separately in an elucidative program.

4.1 The basic approach

An elucidative program relies on relations between the documentation and the program. The relations are represented in the documentation, but presented as links from the documentation to the programs as well as the other way around. The initial conception of Elucidative Programming, and its relations to Literate Programming, is described in a requirements paper [14]. The paper *Elucidative Programming* [13] gives additional descriptions, including details about the original version of the Scheme Elucidator. The Java Elucidator [22] is a tool inspired by the original Scheme Elucidator.

This paper addresses the Scheme Elucidator 2, which uses an XML language as the front-end format, and XHTML (with CSS) in the back-end. The actual documentation can either be written in the special purpose markup language of the original elucidator [13] or by use of an XML documentation language (via LAML expressions in Scheme). The latter approach is recommended, because it is aligned with the approach of SchemeDoc and other XML languages in LAML, but not least because of the power of *programmatic authoring* [15]. In this paper we will stick to documentation authored via the XML language, used via LAML.

The Scheme Elucidator can handle a single documentation file and an arbitrary number of Scheme source files. Together, these files form a *documentation bundle*. In addition, an arbitrary number of SchemeDoc manuals can be taken into account. If a procedure `p`, documented by SchemeDoc, is applied in a program or mentioned in the documentation, there will be links to the interface documentation of `p` from the places where `p` is called or mentioned. In addition, all applications of R4RS/R5RS procedures and syntactic forms are linked the to appropriate locations in the Scheme Report [6].

An elucidative program is presented as a collection of frames in a web browser, using the layout shown in Figure 5. The basic and novel idea related to the presentation of an elucidative program is the *mutual navigation* between the Documentation frame and the Program frame. Given some documentation *d* shown in the Documentation frame, a program fragment described in *d* may be scrolled into view in the Program frame. Symmetrically, given a program abstraction *p* shown in the Program frame, a section of documentation which mentions or explains *p* may be scrolled into

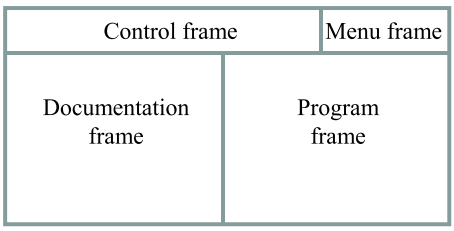


Figure 5: *The Scheme Elucidator frame layout.*

view in the Documentation frame. The Menu frame provides for selection of source programs in a documentation bundle, and the Control frame holds the main navigational icons as well as a structural index of the documentation.

4.2 An example

As a concrete illustration of the Scheme Elucidator 2 we show a small demo of an elucidative Scheme program. The demo includes a single LAML documentation file and two Scheme source files, namely `prog1.scm` from Figure 1 and another program, `prog2.scm`, with a few simple, higher-order Scheme functions. The entire documentation source file is shown in Appendix A. A snapshot of the elucidative program, which makes use of the frame layout shown in Figure 5, can be seen in Figure 6.[1] Notice that only a few links are underlined in the two large frames of Figure 6.

The documentation frame of Figure 6 contains a large number of references to abstractions in `prog1.scm` and `prog2.scm`. There are links from the documentation to the definitions of the Scheme programs. The other way around, the documented definitions in the program frame are decorated with links to the documentation sections with relevant explanations. (These links are anchored in the small icons shown just above the `define` forms). Applied names are linked to their definitions in the Scheme source programs. Reversely, the definitions are, via cross reference tables, linked to the abstractions that apply the definitions. To provide for a natural source-like appearance of the Scheme programs, the links are not underlined, and they are shown in selected dark colors.

The small colored circles, called *source markers*, denote details within a Scheme abstraction. Source markers are used for identification of program details, which are discussed in the documentation. In a Scheme source program the source markers are written as, for instance, '`@a`' in a comment. Pairs of similar source markers (in the documentation and in a source program file) provide for a visual correspondence, but they are also navigatable in both directions.

The elucidative program source in Appendix A shows an `elucidative-front-matters` clause and the documentation intro, sections, and entries in between `(begin-documentation)` and `(end-documentation)`. The `source-files` clause in `elucidative-front-matters` enumerates the Scheme source programs of the documentation bundle and the SchemeDoc files that should be taken into consideration. The `color-scheme` clause defines the background colors which are used to group related source files to each other in a visual way. The `documentation-intro`, `documentation-section`, and `documentation-entry` clauses represent the actual documentation, and they hold the references to the abstractions of the Scheme programs.

Within the documentation it is possible to address a Scheme definition via the name of the definition, both with and without source file qualifications. The XML element mirror functions `weak-prog-ref` and `strong-prog-ref` are used for this purpose. A strong program reference is intended as a reference to a Scheme definition from a context, which explains the definition. A weak program reference is used when a definition is mentioned in other contexts. It should be noticed that a source marker in the documentation is implicitly related to the closest preceding strong program reference. The distinction between weak and strong program references is not always objective.

Due to the many occurrences of weak and strong program references, the author may choose to introduce "flexible abstractions" on top of these, either ordinary Scheme functions or XML-in-LAML abstractions (see Section 2).

The bodies of documentation entries and sections are typically HTML paragraphs. At the most detailed level, textual content is represented as string constants. As a consequence of the LAML parameter passing rules discussed in Section 2, there is white space in between element content items, unless suppressed by the underscore symbol.

4.3 Tool Support

The Scheme Elucidator tool processes a documentation bundle, as defined in Section 4.1. The result of the processing is a collection of HTML files, which can be presented and explored in an Internet browser.

During program development, it is important to support elucidative programming in the programming environment. Without tool support it is difficult and error prone to manage the linking process between the documentation and the abstractions in the source programs. We have developed Emacs tools that support Elucidative Scheme programming. The tools support the creation of links and they make it possible to follow links within the editing environment.

If the programming is done in an integrated development environment (IDE) it is attractive to integrate Elucidative Programming (development as well as browsing) in the IDE. It is a non-trivial task to come up with a good integration. The integration of the Java Elucidator and the TogetherJ IDE shows how this can be done [32].

4.4 Implementation issues

Like the SchemeDoc tool, the Scheme Elucidator is implemented in Scheme. The most challenging aspect of the implementation is the rendering and the linking of the Scheme source programs, i.e., the creation of the program frames. The rendering is done by a simultaneous traversal of the textual Scheme program and the parsed Scheme program. Thus, the Scheme Elucidator processes both the textual and the structural representation of the program. The source program text holds the information about the program layout. The parsed Scheme program makes it convenient to look ahead, for instance into the actual definition following a definition comment. The handling of quotations and quasiquotations calls for particular attention during the traversals, because of differences between the textual and the structural representations.

[1] For better viewing and color presentation please bring up Figure 6 in your own Internet browser using the link on the web resource page [21] of this paper.

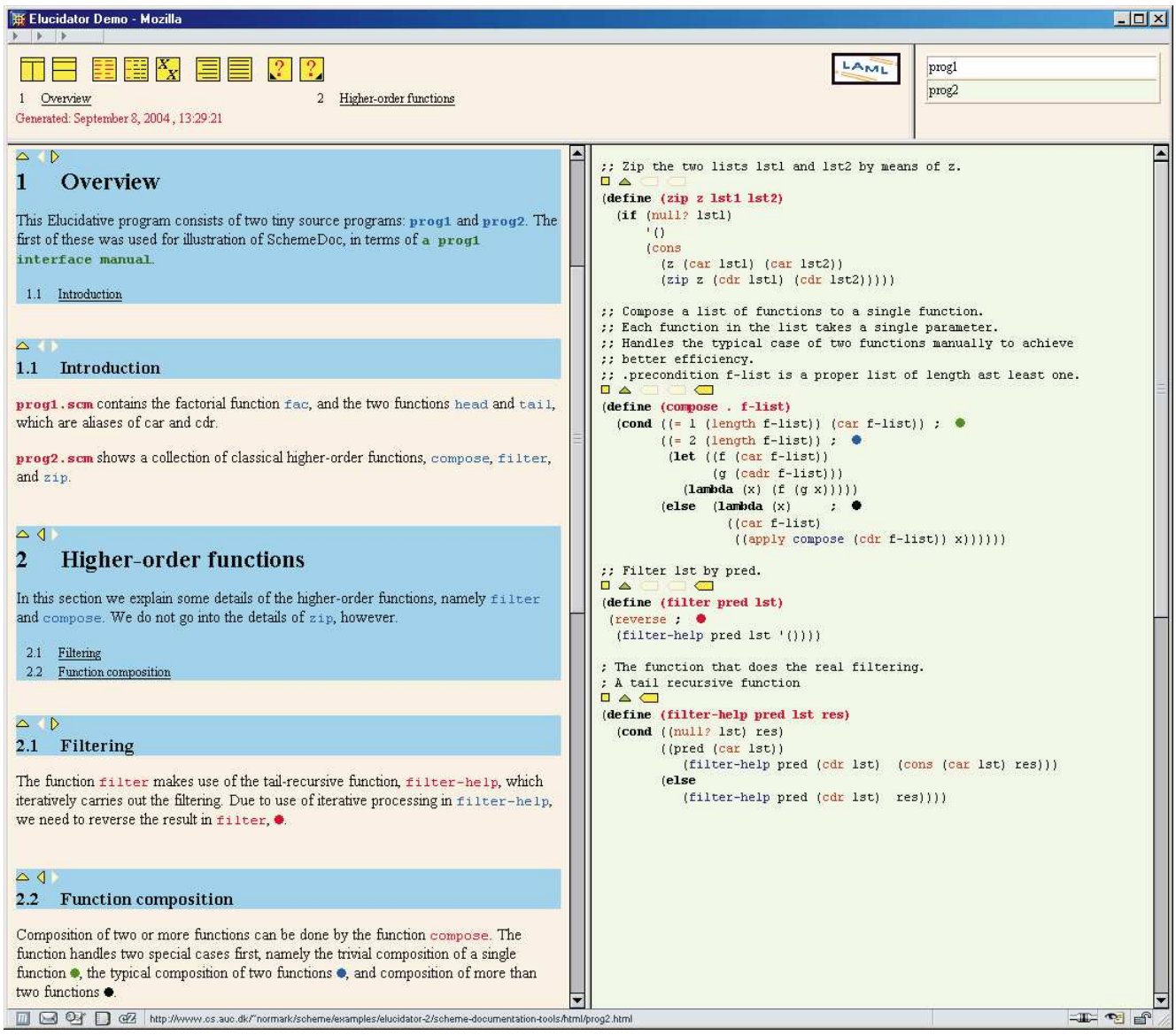


Figure 6: *A snapshot of an elucidative Scheme program.*

The current version of Scheme Elucidator is not aware of name binding effects caused by use of syntactic abstractions (macros). However, the Scheme Elucidator is aware of the syntactic forms that introduce local name bindings (`define`, `lambda`, `let`, `let*`, and `letrec`).

4.5 Limitations and extensions

We have a number of ideas of future improvements of the Scheme Elucidator, some of which remedy weaknesses of the current version of the tool.

As noticed in Section 4.4, the Scheme Elucidator does not expand macros during the processing of Scheme source programs. It implies, for instance, that the Scheme Elucidator does not take definitions into account which are caused by macro expansion. As a remedy, it has been proposed that the Scheme Elucidator can be told about macros that expand into definitions, and how to extract the names defined by applications of such macros [24].

The Scheme Elucidator can refer to a particular version of a Scheme source file, typically the most recent version. During a long program development process it will often be useful to address the way the program is evolving, more specifically the differences between an early version and the current version of a program. We have clearly felt the need for such facilities in the Elucidative Program that documents the Scheme Elucidator itself (accessible via the accompanying web resource page [21]). Due to this reasoning, it would be relevant to include some support of versioning, at least in a way such that an early version of a program source file can be accessed in a flexible way.

The addressing scheme, realized as a relation between the entities of the documentation and the definitions in the source programs, is

not perfect. In the current tool, it is only possible to address top level entities in the source programs. It would be desirable to be able to address local name bindings as well. The main price to be paid for this would be a more complicated addressing mechanism, and a potential additional burden on the documentation writer.

The Scheme Elucidator uses mutual navigation between the Documentation frame and the Program frame (see Figure 5) based on a bidirectional linking scheme. A literate program [9] presents program fragments within sections of the documentation. In a future development of the Scheme Elucidator we wish, as a supplementary means, to be able to extract program fragments from the source program and to inline these in the documentation. Such a facility is already supported by the Java Elucidator [31], and it resembles the extraction idea of L2T [23], which we briefly review in Section 4.6 of this paper.

The Scheme Elucidator supports a single monolithic documentation node, with two levels of sectioning (sections, and subsections which are called entries). As a minor and relatively easy extension, some programmers call for a more general hypertextual structuring of the documentation in multiple nodes. The Java Elucidator [22] supports multiple documentation nodes.

4.6 Similar work

A variety of work has been done for Scheme, which loosely can be categorized under the umbrella of Literate Programming. Most of this work is oriented towards printed output, typically via use of LaTeX.

SchemeWEB [25] is characterized as "a Unix filter that allows you to generate both Lisp and LaTeX code from one source file". As the novel aspect, SchemeWEB is able to identify Scheme (Lisp) expressions in a LaTeX text. A Scheme expression starts with a '(' at the beginning of a line, and it ends with the matching ')'. The text outside Scheme expressions is considered as documentation. The SchemeWEB tool provides for simple weaving, tangling, and untangling in the web sense [10] of these words.

STOL [11] is tool for presenting a Scheme Program as a LaTeX document. STOL is described as a Literate Programming Tool, and it uses specialized markup as well as LaTeX markup in ordinary Scheme comments. During processing, Scheme code is outputted unaltered, whereas the Scheme comments are transformed relative to specialized markup rules. STOL cannot control the orderinging of the program explanations relative to the ordering of the program constituents, and it is therefore somewhat misleading to call it a literate programming tool. STOL is like a SchemeDoc tool which presents the full source code.

L2T (Lisp to Tex) [23] is a literate programming tool created by Christian Queinnec. L2T is able to extract program fragments from Scheme source files and to insert them in a TeX context, which serves as a program essay. L2T allows the source programs and the documentation to be represented separately. Program fragments are extracted and inserted in the TeX document upon preprocessing of the TeX document with the Lisp2TeX tool. L2T has been used extensively by its author (and by others) for books and papers about Scheme programs.

Mole [12] is Kirill Lisovsky's system for analyzing, repositing, and presenting Scheme source programs. Mole recognizes chapters, sections and units of Scheme definitions. The analysis leads to an SXML [8, 7] representation of a Scheme program. A variety of different queries and extracts can easily be made on the basis of the SXML representation. The presentation, which is currently supported by Mole, is targeted at HTML. The presentation makes use of outlining for presentation of the programs and the program comments at various levels of abstraction.

5 Conclusions

A tool with the properties of SchemeDoc is essential for communication of library interfaces. LAML SchemeDoc supports extraction of distinguished documentation comments from Scheme source programs, and presentation of these as HTML documents. The separate SchemeDoc Indexing tool supports the indexing and organization of a set of SchemeDoc manuals in a 3-framed browser. As the most novel contribution, SchemeDoc is able to document XML DTDs. Due to the difficulties of reading many XML DTDs this is a valuable facility in its own right. However, the documentation of XML DTDs is of particular importance for LAML, because XML languages are represented as libraries of Scheme functions in LAML.

There is no common agreement on the conventions, formats, and the markup of documentation comments in Scheme. This has lead to a number of mutually incompatible tools, as discussed in Section 3.5. Based on this observation it might be worthwhile for the Scheme community to come up with a recommended format for documentation comments in Scheme source programs.

Seen from the standpoint of traditional program documentation, and in comparison with SchemeDoc, the Scheme Elucidator is a tool for documentation of internal aspects of a Scheme program. From a more open minded point of view, the Scheme Elucidator is a tool for *program exploration*. The exploration can be done within a single source file, between program source files (following chains of name usages both forward and backward), between the program files and the authored documentation, between a program and SchemeDoc interface documentation, and between the program and the Scheme Reference Manual. We find that the Scheme Elucidator is a valuable contribution whenever there is a need to write about Scheme programs, for tutorial, educational or scientific reasons.

Both LAML SchemeDoc and the Scheme Elucidator are bound to the LAML software package. Both tools make use of particular XML front-end languages, as well as XHTML in the back-end. All involved XML languages are represented as mirrors in Scheme. Due to the LAML connection, both tools can be used on all the platforms and Scheme Systems where LAML is running. Thus, in contrast to many similar tools (see Section 3.5 and 4.6) the tools discussed in this paper are not bound to any particular Scheme system. Whereas some other similar systems, such as Scribe [28], support multiple back-ends (and thus multiple target formats) our documentations tools can only generate HTML files.

LAML SchemeDoc has been indispensable for the documentation of LAML libraries (including the mirrors of XML languages in Scheme). The Scheme Elucidator 2 has been used by the author for writing a comprehensive LAML tutorial [19] which currently consists of seven elucidative program parts. The original Scheme Elucidator has also had external users.[2]

[2] See Anton van Straaten's documentation of "An Executable Implementation of the Denotational Semantics for Scheme" at `http://www.appsolutions.com/SchemeDS/ds.html`.

The Scheme documentation tools discussed in this paper can be downloaded as free software from the LAML home page [18]. The details reflected in this paper pertain to LAML version 25.10.

Acknowledgements

I wish to thank the reviewers for useful comments on the initial version of this paper.

6 References

[1] World Wide Web Consortium. Extensible markup language (XML) 1.0, February 1998. http://www.w3.org/TR/-REC-xml.

[2] Lisa Friendly. The design of distributed hyperlinked programming documentation. In Sylvain Frass, Franca Garzotto, Toms Isakowitz, Jocelyne Nanard, and Marc Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France*, 1995.

[3] Tony Garnock-Jones. Docscm documentation: version 0.1. Available via http://homepages.kcbbs.gen.-nz/∼tonyg/chicken/, September 2002.

[4] Aubrey Jaffer. Schmooz. http://swissnet.ai.mit.-edu/∼jaffer/Docupage/schmooz.html, 2002.

[5] Aubrey Jaffer. SLIB - the portable Scheme library version 2d3. http://www-swiss.ai.mit.edu/∼jaffer/-slib.pdf, 2002.

[6] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[7] Oleg Kiselyov. SXML specification. *Sigplan Notices*, 37(6):52–58, June 2002. Also available from http://-okmij.org/ftp/papers/SXML-paper.pdf.

[8] Oleg Kiselyov and Kirill Lisovsky. XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT. 2002. Presented on International Lisp Conference 2002 (ILC 2002). Available from http://okmij.org/ftp/papers/SXs.pdf.

[9] Donald E. Knuth. Literate programming. *The Computer Journal*, May 1984.

[10] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison Wesley, 1993.

[11] Daniel Kobler and Daniel Hernández. STOL—literate programming in Scheme. *Lisp Pointers*, 5(4):21–30, October-December 1992.

[12] Kirill Lisovsky. Scheme program source code as a semistructured data. In *2nd Workshop on Scheme and Functional Programming*, September 2001. http://kaolin.unice.fr/-Scheme2001/article/lisovsky.ps.

[13] Kurt Nørmark. Elucidative Programming. *Nordic Journal of Computing*, 7(2):87–105, 2000.

[14] Kurt Nørmark. Requirements for an elucidative programming environment. In *Eight International Workshop on Program Comprehension*, pages 119–128. IEEE, June 2000. Also available via [16].

[15] Kurt Nørmark. Programmatic WWW authoring using Scheme and LAML. In *The proceedings of the Eleventh International World Wide Web Conference - The web engineering track*, May 2002. ISBN 1-880672-20-0. Available from http://-www2002.org/CDROM/alternate/296/.

[16] Kurt Nørmark. The Elucidative Programming home page, 2003. http://www.cs.auc.dk/∼normark/-elucidative-programming/.

[17] Kurt Nørmark. Web programming in Scheme with LAML. To appear in Journal of Functional Programming, April 2003. Available via [18].

[18] Kurt Nørmark. The LAML home page, 2004. http://www.-cs.auc.dk/∼normark/laml/.

[19] Kurt Nørmark. The LAML tutorial. Part of the LAML system, April 2004. Available via [18].

[20] Kurt Nørmark. The SchemeDoc home page, 2004. http:-//www.cs.auc.dk/∼normark/schemedoc/.

[21] Kurt Nørmark. Web resources of the current paper, August 2004. http://www.cs.auc.dk/-∼normark/scheme/examples/elucidator-2/-scheme-documentation-tools.

[22] Kurt Nørmark, Max Rydahl Andersen, Claus Nyhus Christensen, Vathanan Kumar, Søren Staun-Pedersen, and Kristian Lykkegaard Sørensen. Elucidative programming in Java. In *The Proceedings on the eighteenth annual international conference on Computer documentation (SIGDOC)*. ACM, September 2000.

[23] Christian Queinnec. L2T: a literate programming tool. Available via http://www-spi.lip6.fr/∼queinnec/-WWW/l2t.html.

[24] Matthias Radestock. Use of the Scheme Elucidator with SISC. personal correspondence, July 2003.

[25] John Ramsell. SchemeWEB. http://www.tug.org/-tex-archive/web/schemeweb/.

[26] Schematics SchemeDoc. http://schematics.-sourceforge.net/schemedoc.html.

[27] A Scheme documentation generator. Contained in ftp://-ftp-sop.inria.fr/mimosa/fp/Bigloo/contribs/-scmdoc.tar.gz, 1998.

[28] Manuel Serrano and Erick Gallesio. This is Scribe! Presented at the 'Third Workshop on Scheme and Functional Programming', October 2002. http://www-sop.inria.fr/-mimosa/fp/Scribe/doc/scribe.html.

[29] Sun Microsystems. Javadoc tool home page (sun microsystems). Available from http://java.sun.com/products/-jdk/javadoc/index.html, 2004.

[30] Dimitri van Heesch. Doxygen. http://www.doxygen.org, 2004.

[31] Thomas Vestdam. Generating consistent program tutorials. In *Proceedings of NWPER'2002 - Nordic Workshop on on Programming and Software Development Tools and Techniques*, 2002. Available via http://dopu.cs.auc.dk/publications/.

[32] Thomas Vestdam. Elucidative Programming in open integrated development environments for Java. In *Proceedings of the 2nd International Conference on the Principles and Practice of Programming in Java*, pages 49–54, June 2003. Available via http://dopu.cs.auc.dk/publications/.

A The elucidative program source

In this appendix we show the LAML source of the elucidative demo program, which we discussed in Section 4, and illustrated in Figure 6.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "xml-in-laml/elucidator/elucidator")

(elucidator-front-matters

 'laml-resource "true"
 'scheme-report-version "r5rs"

 ; OVERALL attributes
 'table-of-contents "shallow" ; detailed or shallow
 'shallow-table-of-contents-columns "3"
 'detailed-table-of-contents-columns "2"
 'source-marker-presentation "image"  ; image, text, colored-text
 'source-marker-char "@"
 'browser-pixel-width "1100"
 'control-frame-pixel-height "120"

 ; INDEX attributes
 'cross-reference-index "aggregated"  ; per-letter, aggregated
 'defined-name-index "aggregated"     ; per-letter, aggregated

 ; PROGRAM attributes
 'initial-program-frame "blank" ; blank, first-source-file
 'large-font-source-file  "true"
 'small-font-source-file  "true"
 'default-source-file-font-size  "small"   ; small or large
 'program-menu "separate-frame"   ; inline-table, none, separate-frame
 'processing-mode "verbose"

 (color-scheme
   (color-entry 'group "doc" (predefined-color "documentation-background-color"))
   (color-entry 'group "index" (predefined-color "documentation-background-color"))
   (color-entry 'group "core" (predefined-color "program-background-color-1"))
   (color-entry 'group "others" (predefined-color "program-background-color-2"))
 )

 (source-files
   (program-source 'key "prog1"
                   'file-path "../../manual-xml-in-laml/scheme-documentation-tools/prog1.scm"
                   'group "core" 'process "true")
   (program-source 'key "prog2" 'file-path "src/prog2.scm"
                   'group "others" 'process "true")

   (manual-source  'key "laml-lib"
                   'file-path "../../../lib/man/general"
                    'url "../../../lib/man/general.html")
   (manual-source  'key "prog1-man"
                   'file-path "../../manual-xml-in-laml/scheme-documentation-tools/man/prog1"
                   'url "../../../manual-xml-in-laml/scheme-documentation-tools/man/prog1.html")
 )
)

(begin-documentation)

 (documentation-intro
    (doc-title "Elucidator Demo")
    (doc-author "Kurt Normark")
    (doc-affiliation "Aalborg University, Denmark")
    (doc-email "normark@cs.auc.dk")
    (doc-abstract
      (p "This is a brief demo example of an Elucidative Program")))
```

```
(documentation-section
  'id "overview-sect"
  (section-title "Overview")
  (section-body
    (p "This Elucidative program consists of two tiny source programs:"
       (weak-prog-ref 'file "prog1")"and" (weak-prog-ref 'file "prog2") _ "."
       "The first of these was used for illustration of SchemeDoc, in terms of"
       (weak-prog-ref 'file "prog1-man" "a prog1 interface manual")_ "." )
  )
)

(documentation-entry
  'id "intro"
  (entry-title "Introduction")
  (entry-body
    (p (strong-prog-ref 'file "prog1" "prog1.scm") "contains the factorial function"
       (weak-prog-ref 'name "fac")_ "," "and the two functions" (weak-prog-ref 'name "head") "and"
       (weak-prog-ref 'name "tail")_ "," "which are aliases of" (weak-prog-ref 'name "car") "and"
       (weak-prog-ref 'name "cdr")_ "." )

    (p (strong-prog-ref 'file "prog2" "prog2.scm")
       "shows a collection of classical higher-order functions," (weak-prog-ref 'name "compose") _
       "," (weak-prog-ref 'name "filter") _ "," "and" (weak-prog-ref 'name "zip") _ "." )
   )
)

(documentation-section
  'id "higher-order-sec"
  (section-title "Higher-order functions")
  (section-body
    (p "In this section we explain some details of the higher-order functions, namely"
       (weak-prog-ref 'name "filter")"and" (weak-prog-ref 'name "compose") _ "."
       "We do not go into the details of" (weak-prog-ref 'name "zip") _ "," "however." )
  )
)

(documentation-entry
  'id "filtering"
  (entry-title "Filtering")
  (entry-body
   (p "The function" (strong-prog-ref 'name "filter") "makes use of the tail-recursive function,"
      (strong-prog-ref 'name "filter-help")_ ","
      "which iteratively carries out the filtering. Due to use of iterative processing in"
      (weak-prog-ref 'name "filter-help")_ "," "we need to reverse the result in" (strong-prog-ref 'name "filter") _ ","
      (source-marker 'name "a")_ "." )
  )
)

(documentation-entry
  'id "composing"
  (entry-title "Function composition")
  (entry-body
   (p "Composition of two or more functions can be done by the function"
      (strong-prog-ref 'name "compose")_ "."
      "The function handles two special cases first, namely the trivial composition of a single
       function" (source-marker 'name "b") _ ","
      "the typical composition of two functions" (source-marker 'name "c") _ ","
      "and composition of more than two functions" (source-marker 'name "d") _ "." )
  )
)

(end-documentation)
```

A Framework for Memory-Management Experimentation

Stephen P. Carl
Department of Mathematics and Computer Science
The University of the South, Sewanee, Tennessee

Abstract

Phobos is a framework for experimenting with memory management systems. This framework provides two types of operation – profiling program allocation behavior and simulating the actions of memory management systems. Profiling is used to generate data about a program's allocation behavior including total memory allocation and memory object lifetimes. Simulation is used to measure the performance of different memory management strategies on particular program runs. In both cases, Phobos takes its input from a *trace file* generated during execution of a targeted application which lists the memory events of interest.

This paper describes the design of the Phobos system. In particular, it shows how the system takes advantage of the code structuring facilities provided by PLT Scheme, highlighting the use of signed units, mixin classes, and other features of this system.

1 Introduction

We are developing Phobos, a framework for studying memory management systems. Most popular functional languages, such as Scheme [15], and object-oriented languages, such as Java [1], use some form of *garbage collection* to implement automatic memory management [13]. While there are a number of garbage collection algorithms, most systems today have some form of *generational* collection available. Some languages best known for scripting capabilities, such as Perl [24] and Python [23], use *reference counting* systems for automatic memory management, while implementations such as Jython [14] benefit from advances in the Java language runtime. Our goals for Phobos include classifying the memory allocation patterns of different types of applications and determining how well or poorly different memory management algorithms interact with these patterns.

The framework has two modes of operation: *profiling*, which can be used for studying the allocation behavior of applications, and *simulation*, which can be used to determine how well different garbage collection algorithms perform when matched with applications exhibiting these behaviors. Information about an application's use of memory comes from *memory trace files*, which contain information about object allocations, object deallocations (for profiling), pointer stores and pointer reads (for simulation), and so on. Trace files are produced by instrumented virtual machines or interpreters which log events of interest as they occur.

This paper describes the design of Phobos and simple examples of how it is used. The framework design is based on the program structuring features provided by PLT MzScheme, a R^5RS-compliant Scheme implementation featuring a number of useful extensions including a fully integrated module system, units for creating separately-compilable components, and a Java-like class system which supports *mixin-based* programming [5]. We describe how use of these language features helped create a system capable of specifying various simulator configurations from information provided by the user.

2 Simulator Design

2.1 Design Goals

Phobos has been designed with the following goals in mind:

- **Experimental Control.** The user controls the system through a script for specifying experimental parameters such as heap size, input format, memory management components used in simulation, and statistics to collect.
- **Prototype Development.** Implementations of new memory managers can be prototyped to quickly explore the design space.
- **Language independence.** While initial experiments are targeted at Java programs, other languages can be accomodated by providing an execution environment instrumented to produce trace files.
- **Extensibility.** Phobos can be extended to handle new types of trace file formats, new memory management algorithms, and new types of statistics to collect.

2.2 Structuring the System

The framework is divided into two sets of modules: the *engine* and the *simulation components*. The engine is made up of the main driver, handlers for reading trace files, and an interface for experiment scripts. Simulation components are defined as MzScheme classes and represent the heap and heap partitions, basic memory

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

management systems, and statistics generators.

The engine and components are combined according to information in an experiment script which is written by the user to control the simulator. Information from the script tells the system how to combine the main driver with the functions which read a specific trace file format. The script further specifies the memory management functionality to be used; the script interface determines which classes from the simulation components are needed and loads them at startup time. The memory manager is made by combining classes encapsulated in *units* that represent allocation and collection systems with the heap classes. A detailed description of how and why units are used in our framework is given in Section 5.

Trace handlers are functions which deal with specific memory trace events given in a trace file. The framework currently supports three formats, one for manual memory management, one for a JVMPI-compliant profiling agent for Java programs [22], and one for general simulation. New formats can be accomodated by developing a new set of handler implementations. As each memory event is read from the trace file, the driver transfers control to the appropriate handler function for checking that the event is in the proper format and creating an object that represents the trace. Such objects in turn cause some change in the heap by sending it a message corresponding to the event type (e.g., object allocation, object deallocation, pointer store, etc.).

When profiling, the heap is used to simply store allocated objects, then remove them when deallocated, while maintaining a set of counters which track basic statistics. When doing simulation, the heap object sends messages to one or more *heap frames* which it manages. A heap frame is conceptually just a range of addresses in (simulated) memory, coupled with a specific set of methods for handling allocation or collection; these impose a logical organization on the heap frame. This organization allows us to model monolithic heaps which use a single allocator and collector, and also heaps that are partitioned into regions which are managed differently, such as in generational garbage collectors.

The simulation components that report statistics log basic information in each memory object allocated. The system produces two files of raw data. The **log file** contains allocation time,[1] size, and deallocation time for each object allocated. The **lifetimes file** contains the *lifetimes* of each object, where lifetime is the difference between deallocation time and allocation time. The information in these files is suitable for processing by external tools such as Matlab for producing graphs, such as object type and lifetime distributions.

3 Profiling Applications

The simplest use of the system is to profile the memory usage of an application. A trace file to be used for profiling records the actions of the memory manager used by the system which executes the application, including GC start and stop events, per-object allocation and deallocation events, and object copy events. In profile mode, Phobos simply replays these events in a *shadow heap* and tracks information about each object and the events which affect them. It also computes statistics about each object when they are deallocated or the trace file ends, and displays the amount of heap space needed by the application being profiled, the total amount of allocation both in number of objects allocated and size of memory used, and the number of garbage collections required including the number of objects and total memory collected at each.

The following simple script runs a single profiling experiment:

```
(experiment
 (connect 'PROFILE "~/traces/robo-trace"))
```

The `connect` form specifies the trace format type and the full pathname to the trace file. When an experiment run begins, the system selects the trace handlers to use based on the trace format type (though this can be overridden in the script) and combines these with the main driver and the simulation components used for profiling. When invoked, the driver first attempts to open the trace file specified, and then calls the trace handler functions to read and respond to memory events stored in the trace file.

Once the trace file processing is completed, the system displays the global heap statistics it has computed:

```
For this trace file run:
Total types recorded: 1022
Total amount of allocation: 98993720 bytes
Total number of objects allocated: 2350147
Total number of objects collected: 2237897

4786686 Events Processed
```

Also, the statistics component produces the object lifetimes and allocation log.

4 Simulating Memory Managers

In simulation, the trace file produced when an application is run contains allocation events, pointer update events, and enough information about the execution to drive the actions of a simulated memory management system. A typical simulation script describes each experiment to be run, including the trace file to be used, the memory management system (or systems) to simulate, and the characteristics of the simulated heap.

For example, the following script runs a single experiment using a trace file `robo-trace` on a typical heap managed by a best-fit allocator and mark/sweep garbage collector:

```
(experiment
   (connect SIM "~/traces/robo-trace")
   (base-heap 64 0 (allocator first-fit)
                   (collector mark-sweep)))
```

The simulation is again driven by the memory trace file in the path specified in the `connect` form. The second part of the experiment script specifies the characteristics of the heap: size is 64 Mbytes, the base of the heap (conceptually) starts at address 0, and it is managed by a first-fit allocator and a mark/sweep garbage collector. This script runs the experiment and reports the statistics just as in the profiler.

The names `first-fit` and `mark-sweep` are predefined. In general, the names are used to choose the module in which the particular class definition implementing these algorithms is found. For example, `first-fit` is defined in a module in the file **first-fit.scm**, which provides a class that implements the first-fit allocation policy. Actually, the algorithms are defined as *mixin classes*, and what is provided is a function which creates a new class representing a

[1] Per the GC literature, allocation time is measured in bytes allocated so far – the first object is allocated at time 0, the next is allocated at time 0 + (size of first object), and so on.

heap frame extended with the mixins. Section 6 describes this in more detail.

Experimental setups can be defined and saved singly or in groups. Saving an experimental setup by associating it with a name allows results to be labeled with experiment names rather than with experiment characteristics. In the example shown in Figure 1 we define a set of experiments, called an experiment *suite*, which will be run one after the other by the simulator. The suite is defined by the form `define-experiment-suite`.

5 Under the Hood

In MzScheme, a *unit* is a separately compilable component which can be linked to other units to create a (as yet unevaluated) program. Units may import external variables which are used in the body of the unit, and may export its own variables to be imported by other units. When the list of variables imported or exported is long, *signed units* are used instead as a convenience. In this case, the programmer provides *signatures* to specify those names to be exported to other units, and import using other unit's signatures. The program formed by linking units is evaluated when *invoked* with the `invoke-unit` (with regular units) or `invoke-unit/sig` forms (with signed units).

In Phobos, evaluating an experiment script selects those units whose code is to be used to handle specific types of traces and construct the simulated heap. Scripts are processed by passing the script file name on the MzScheme command line. The script is then evaluated as Scheme code, using the definitions provided by Phobos. The `experiment` form is a macro which uses the information in its body forms to select the appropriate units (representing the engine and required components) and link them together to create a *compound unit* at runtime. Invoking this compound unit starts the main driver.

The simple profiling experiment script shown in Section 3 elaborates to the `let*` form shown in Figure 2. This code creates and invokes a compound-unit out of a set of signed units; the signatures (defined elsewhere) are given by the symbols which end with a caret (`^`). The compound-unit is the result of linking the individual units `htprof-handlers@`, `unit-heap@`, `sim-driver@`, and `exp@` together. The first three of these are loaded into the system at runtime by a form (elided in the figure) called `dynamic-require`.

The unit `exp@`, which comes first in the `let*` form, defines the "command line" for the engine, using parameters from the script. The `compound-unit` form then links the units which define the specific trace handlers used by engine together with the simulated heap, the driver unit, and the unit which defines the command line. The driver unit exports the procedure name `sim-driver` used by the new unit `exp@`. The link step returns the compound unit `prg@` that is invoked in the `let*`. Invoking the compound unit has the effect of calling `sim-driver` and running the simulation in profiler mode. The results produced will be labeled with the experimental characteristics, that is, the name of the trace file, the memory manager used, and the heap size and layout.

6 Structure of the Simulated Heap

The simulated heap is made up of one or more heap frame objects. For modeling monolithic heaps one heap frame is sufficient. However, in modern runtime systems heaps tend to be partitioned. Generational garbage collectors partition by age; other recently proposed systems partition by type [17] or by connectivity [9]. To model these systems, each heap frame represents a different *partition* of the heap. Global attributes of the heap are captured by the `heap%` class, which also holds the first heap frame. The structure of the definition (minus method code) is as follows:

```
(define heap%
  (class* object% (heap<%>)
    (init-field
     ;; an object that collects statistics
     stats
     ;; default heap size is 32 Mbyte
     (initial-size (expt 2 25))
     (max-size initial-size)
     (alloc-frame
       (make-object heap-frame% initial-size)))

    (field (bytes-allocated 0))
    (field (total-allocated 0))
    (field (total-objects 0))

    (define/public (allocate trace)
      ;; updates global properties of the heap
      ;; sends allocate message to alloc-frame
     ...)

    (define/public (deallocate object-id)
      ;; updates global properties of the heap
     ...)
     ...)))
```

This definition creates the class `heap%` consisting of a set of fields (three defined by the `init-field` form and four by the `field` forms) along with a set of methods (only a subset of the class methods are shown). The field `alloc-frame` refers to the initial heap frame. Each heap frame refers to the "next" heap frame in the system. For example, to define a semi-space copying collector, two frames are used, each refering to the other. For generational collectors, each generation is a separate heap frame which each refer to the succeeding generation in the system.

Each heap frame is defined by a class `heap-frame%` which includes methods for allocating blocks, collecting unreachable objects, and handling pointer reads and writes as shown:

```
(define (heap-frame% %)
    (class* % (heap-wrapper<%>)
      (inherit store! lookup remove!)
      (rename (super-terminate terminate))
      (init-field
        initial-size
        (next-frame '()))
      (field (frame-bytes-allocated 0))
      (field (start-addr 0))
      (field (end-addr (- initial-size 1)))
      (field (roots '()))

      ;; Method Declarations
      (define/public (allocate-slot obj size)
       ;; allocates the next available block
       ;; large enough to store obj of given size
        ...)

      (define/public (collect-slots)
        ;; dummy collector
```

```
(define-experiment-suite gc-suite
   "run experiments on two tracefiles"
   (experiment
     (connect SIM "~/traces/robo-trace")
     (base-heap 32 0 (allocator first-fit)
                     (collector mark-sweep)))

   (experiment
     (connect SIM "~/traces/kaffe-trace")
     (base-heap 32 0
       (partition (name nursery 16)
                  bump-pointer
                  (copy-promote (partition (name old 16) best-fit mark-sweep))))))

(simulate gc-suite)  ;; kicks off experiments
```

Figure 1.

```
(let* ((exp@ (unit/sig () (import sim-driver)
               (sim-driver "/traces/robo-trace" JVMPI (expt 2 24))))
       (prg@ (compound-unit/sig (import) (link [HANDLE : trace-handlers^ htprof-handlers@]
                                               [SIMHEAP: unit-heap^ unit-heap@]
                                               [DRIVER : sim-driver^ (sim-driver@ HANDLE SIMHEAP)]
                                               [RUN    : () (exp@ (DRIVER sim-driver))])
                                     (export))))
  (invoke-unit/sig prg@))
```

Figure 2. Elaboration of `experiment` form

```
       ...)

      (define/public (read addr)
        ;; pointer read
        ...)

      (define/public (write addr ptr)
        ;; pointer write
        ...)))
```

The default heap frame object implements the `NoGC` storage manager, which creates new objects in the next available chunk of memory and removes objects without making the newly-freed space available for future allocations. More useful allocator and collector mechanisms are provided in the form of *mixin classes* which extend `heap-frame%` by overriding the `allocate-slot` and `collect-slots` methods. The `read` and `write` functions can also be overridden for implementing read or write barriers as needed. When a specific type of heap manager is chosen for simulation, the heap frames are created by choosing appropriate allocator and collector subclasses, creating extensions by mixing these in, and then instantiating the resulting classes.

This organization is accomplished as follows: a *mixin* is created in MzScheme by defining a class whose superclass is specified as a parameter, using the `define` form. For example:

```
(define (make-mixin super-class)
  (class super-class ...extension...))
```

The actual class is created by calling the resulting procedure and passing in the name of the superclass to be extended.

There are two main benefits of using mixin classes in this system. First, allocators and collectors can be combined independently as long as they are compatible (for instance, the system will generate an error when processing a script which pairs a non-moving allocator with a copying collector). Second, when placed in their own units, mixin extensions can be selected and combined to form a single unit representing the simulated heap by importing the actual superclass at link time.[2] This allows us to essentially create different heap frame classes on the fly, combining them to form a multiple-partition heap where each partition is managed using a different strategy.

An allocation event only affects a single heap frame. Each allocation algorithm is defined in a separate unit as a mixin class which contains at least the method `allocate-slot` as shown in this example:

```
(define (bump-pointer super%)
  (class super%
    (init-field
      size
      (pointer 0))
    (define/override (allocate-slot trace)
      ;; defines new allocator
     ...)))
```

The procedure `bump-pointer` takes an argument `super%` which is the superclass of the mixin. The `allocate-slot` method overrides that defined in the superclass. This mixin defines the "bump pointer" allocator (also known as fast allocation) which reserves the next free address in the heap frame for the object being allocated. The `init-field` form defines two fields; `size` is the maximum size of the heap frame, and `pointer` tracks the next available position in the frame.

Collectors are created in the same way. A particular collector component overrides the method `collect-slots` and can include any

[2] More information on the use of units and mixins in MzScheme can be found in Findler and Flatt's ICFP'98 paper [4].

other supporting methods or fields necessary. The collector components will be combined with some superclass (again, not usually known in advance) using the same mixin style as with allocators. In general, the `collect-slots` method is called by the allocator when there is no more space available in the heap frame or some threshold size is reached.

Once defined, units for allocators and collectors become part of a library of components to be used in experiments. The name of the module which defines a specific component is given in the experiment script which selects the proper units and evaluates the procedures for each mixin class. When evaluated, these procedures generate a new class which will extend either `heap-frame%` or some subclass of it. The actual superclass does not have to be known in advance. In this way, classes representing the simulated heap are created on the fly based on the experiment script.

7 Memory Management Components

In this section we cover some of the forms used in Phobos experiment scripts to generate the memory management classes. New components are being added as the system matures. Components are in general added by writing mixin classes built along the same lines as `bump-pointer`. In more advanced cases, new macro forms may be required.

7.1 Allocators

The `allocator` form specifies the unit to be used for allocation. Examples of allocators currently available include the default `bump-pointer`, simple `first-fit` allocation, and the more advanced `seg-freelist` for implementing a segregated freelist allocator. An allocator is specific to a single heap frame.

7.2 Collectors

The `collector` form specifies a unit to be used for garbage collection. Collectors defined using this form are generally used to manage a single partition in the heap. The `mark-sweep` and `mark-compact` collectors are two examples of collectors which can be used with this form.

7.3 Partitioned Heaps

The form `partition` allows the user to define the way the heap is divided into heap frames, usually for copying collectors. This form has the following structure:

```
(partition (name <identifier> <size>)
           <allocator>
           <collector>)
```

The `name` subform is optional and associates an identifier with the partition as well as its size (in Mbytes). If `name` is not used, the size is specified there instead. The `<allocator>` and `<collector>` parameters are either the unit names as described before, or one of `copy-to` or `copy-promote`, each of which specifies copy collection between partitions.

The form of `copy-to` is `(copy-to` *size*`)`, where *size* gives the size of a second partition, or semispace. This form is used to create a two-semispace copy collector. Elaboration of this form generates two heap frames which refer to each other via the `next-frame` field.

The form of `copy-promote` is `(copy-promote` *partition-form*`)`, where *partition-form* is another partition declaration. This is generally used to create a generational memory manager, though provision for promoting based on criteria other than age is planned. The second partition form is the "older" generation which receives copies of objects which survive collections of the original partition. Note that partitions defined in `copy-promote` can themselves declare `copy-promote` as their collector (and so on) to generate more than two generations. Currently the form uses a default remembered set write barrier to catch intergenerational pointers.

8 Future Work

8.1 Instrumenting New Implementations

The Phobos framework was originally conceived as a tool to gauge the allocation characteristics of functional languages designed to compile to the Java Virtual Machine [20]. This is one reason why our current set of trace files are generated by executing Java programs. In the future, we would like to conduct experiments with trace files generated from progams executed directly by implementations of Scheme and other languages. This will require modifying existing execution environments to generate information about the memory events of interest.

The Garbage Collection website [12] includes a small repository of memory traces, which is intended to eventually represent many traces from applications written in different languages. The research community has apparently been slow to contribute to this repository; we would like to contribute to it soon and encourage other researchers to do so.

8.2 Visualization

Currently all visualization of data generated by Phobos is done using Matlab to generate graphs. It would be nice to have a set of tools for presenting interesting views of the allocation behavior of the programs and the performance of the memory managers. We will be evaluating other tools specifically aimed at graphing (such as PLTplot [6]), and profiling (such as EVOLVE [25]).

8.3 Developing New Managers

The memory manager components defined in Phobos are useful for studying the behavior of commonly used systems. The scripting system needs to be more flexible, however, if newly proposed and researched systems are to be implemented using this approach. In particular, the `copy-promote` form needs to be modified or complemented so that alternative write-barriers can be specified as well as different criterion for promoting objects.

Researchers who develop improved memory managers may want to develop prototypes to study their high-level performance on application traces. While writing allocators and collectors in Scheme can be an enjoyable exercise, we would like to develop memory management components in an *embedded language* built to work directly with the experiment scripting facility. We would like to try to develop such a "little language" [16] to aid in the process of building up the framework. The more allocators and collectors added to the library of components, the more experience we will have to better understand the abstractions and interfaces that the language must support.

9 Related Work

Simulators are used to study both the allocation behavior of specific applications and the behavior of memory management techniques. One of the first described systems was MARS, the Memory Allocation Research Simulator, described in Ben Zorn's dissertation [26]. This simulator was attached to a running LISP system and allowed the user to study the impact of using different (simulated) garbage collection algorithms with a set of applications. Zorn also proposed using a language specific to this domain for describing management systems to be simulated, but did not define one himself. To our knowledge this has not yet been attempted.

Simulation is also a key component in the work of Darkovic [19], who studied age-based (generational) collectors in the context of Smalltalk and Java, and Hansen [8], who studied older-first generational collectors in the context of Scheme.

Hölzle and Dieckmann developed a trace-driven simulator to provide data about the memory behavior of Java programs to the garbage collection research community [3]. The system was driven by memory traces generated from applications in the SPECjava98 benchmark suite. The simulator generated data for computing statistical information about object lifetime distributions, size variations, and the amount of heap space required to run each program. Few if any such simulators have been made publically available to the research community.

In the Jikes^{TM}Research Virtual Machine, new memory management mechanisms can be implemented directly (not simulated) by subclassing a set of provided Java GC classes which provide the base garbage collector. This allows the programmer to experiment with and determine the effects of different managers on an application or set of applications directly, without the need for generating trace files. However, the entire virtual machine must be rebuilt (a lengthy process) before testing a new manager [10].

Beltway is a framework built on top of Jikes which generalizes copying garbage collection, such that each of semispace, generational, and older-first collection schemes can be defined in a common framework [2]. The simulated heap is divided into some number of partitions called *belts*. Each belt is made up of a number of *increments*; an increment is the unit of allocation. Varying the size and number of belts allows the user to construct any existing copying collector, or create entirely new ones. Furthermore, the system supports partitioning objects in the heap by size, type, or call-site, so several different object characteristics can be exploited at once.

10 Conclusion

This paper has described the design of a framework for profiling the memory allocation behavior of applications and simulating memory-management systems. The framework uses program-structuring features provided by PLT Scheme to build representations of the simulated heap from components chosen by an experimenter at runtime. The use of units to compartmentalize code, specify import and exports in a disciplined way, and link components at runtime makes it possible to specialize the system based on an experiment script.

This approach has given us the ability to build simulations for a number of popular memory managers. It is not clear, however, that building components in this way will be useful for alternative designs currently being researched. It may turn out that some designs may be more difficult to render given the high level of abstraction represented by signed units and mixin classes. But for systems implemented thus far, the approach has allowed us the flexibility of developing components for different allocation and collection mechanisms and make them available to the simulator.

11 References

[1] Ken Arnold and James Gosling. *The Java^{TM} Programming Language, 2nd Edition.* Addison-Wesley, 1998.

[2] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting Around Garbage Collection Gridlock. Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation. In *SIGPLAN Notices.* Vol. 37, No. 5, pp. 153–164, May 2002.

[3] Sylvia Dieckmann and Urs Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. Technical Report 1998-33, UCSB Computer Science Department. December, 1998.

[4] Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. Proceedings of the International Conference on Functional Programming (ICFP '98). In *SIGPLAN Notices*, Vol. 34, No. 1, January 1999.

[5] Matthew Flatt. *Programming Languages for Reusable Software Components*. Ph.D. thesis, Rice University, June 1999.

[6] Alexander Friedman and Jamie Raymond. PLoT Scheme. Fourth Workshop on Scheme and Functional Programming, November 7, 2003, Boston, MA.

[7] Richard P. Gabriel, *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, 1985.

[8] Lars T. Hansen. *Older-first garbage collection in practice*. Ph.D. thesis, Northeastern University, November 2000.

[9] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *The 2002 International Symposium on Memory Management (ISMM 2002)*, pp. 36–49, Berlin, Germany, June 2002. ACM Press.

[10] Jikes Research Virtual Machine from IBM. `http://www.ibm.com/developerworks/oss/jikesrvm`.

[11] Richard Jones' Garbage Collection Bibliography. `http://www.cs.kent.ac.uk/people/staff/rej/gcbib`.

[12] Richard Jones' Garbage Collection Pages. `http://www.cs.kent.ac.uk/people/staff/rej/mtf/traces`.

[13] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[14] Jython Homepage. `http://www.jython.org/`

[15] R. Kelsey, W. Clinger, and J. Rees (Eds). The Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, Vol. 33, No. 9, September 1998.

[16] Olin Shivers. A universal scripting framework, or Lambda: the ultimate "little language." In *Concurrency and Parallelism, Programming, Networking, and Security,* Lecture Notes in Computer Science #1179, pages 254–265, Editors Joxan Jaffar and Roland H. C. Yap, 1996, Springer.

[17] Y. Shuf, M. Gupta, R. Bordawekar, and J.P. Singh. Exploiting prolific types for memory management and optimizations. Proceedings of the 2002 SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02). In *SIGPLAN Notices*, Vol. 37, No. 1, pp. 295–306, January 2002.

[18] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation, Release 1.0*. August 1998. `http://www.spec.org/osg/jvm98/jvm98/doc/index.html`.

[19] Darko Stefanovic. Properties of Age-based Memory Reclamation Algorithms. Ph.D. thesis, University of Massachusetts, February 1999.

[20] Robert Tolksdorf. Languages for the Java VM. `http://www.robert-tolksdorf.de/vmlanguages.html`.

[21] Sun Microsystems Inc. The HotSpot^{TM} Performance Engine. `http://java.sun.com/products/hotspot`.

[22] Sun Microsystems Inc. Java Virtual Machine Profiler Interface (JVMPI). `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html`.

[23] Guido von Rossum. *Python Tutorial*. `http://www.python.org/doc/current/tut`.

[24] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl, 2nd edition.* O'Reilly & Associates, Inc., 1996.

[25] Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren and Clark Verbrugge. EVolve, an Open Extensible Software Visualization Framework. Sable Technical Report SABLE-TR-2002-12. McGill University, School of Computer Science, 2002.

[26] Benjamin Zorn. *Comparitive Performance Evaluation of Garbage Collection Algorithms*. Published as CSD-89-544 from University of California, Berkeley.

trx: Regular-tree expressions, now in Scheme

Ilya Bagrak
University of California, Berkeley
ibagrak@eecs.berkeley.edu

Olin Shivers
College of Computing
Georgia Institute of Technology
shivers@cc.gatech.edu

Abstract

Regular-tree expressions are to semi-structured data, such as XML and Lisp s-expressions, what standard regular expressions are to strings: a powerful "chainsaw" for describing, searching and transforming structure in large data sets. We have designed and implemented a little language, *trx*, for defining regular-tree patterns. We discuss the design of *trx*, its underlying mathematical formalisation with various kinds of tree automata, and its implementation technology. One of the attractions of *trx* is that, rather than being a complete, *ad hoc* language for computing with trees, it is instead embedded within Scheme by means of the Scheme macro system. The features of the design are demonstrated with multiple motivating examples. The resulting system is of general use to programmers who wish to operate on tree-structured data in Scheme.

1 LCD data representations

Semi-structured and tree-structured data has become an important topic in the world of software engineering in the past few years, due to the widespread adoption of XML as a generic representation format for data. While this may be news to rest of the world, it is a very familiar picture to programmers in the Lisp family of languages. The Scheme and Lisp community has long been aware of the benefits of fixing on a general-purpose data structure for representing trees, and specifying a standard concrete character representation for these trees. Lisp s-expressions are essentially XML trees; the Lisp community has worked within the s-expression framework for representing data since the inception of the language in the 1960's.

Part of the power of the Lisp family of languages comes from this focus on s-expressions as the central data structure of the language. A Perlis aphorism [16] captures the benefit well: "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures." S-expressions are the "least common denominator" (LCD) representation for data in the Lisp family of programming languages, in the same sense that strings are the LCD representation in the world of Unix tools: the multitude of functions that operate upon and produce results in this form can therefore easily be connected together to construct larger computations, providing for a large degree of code reuse. In the world of Scheme programming, s-expressions are the universal interchange format.

The charm of s-expressions as an LCD representation is that, unlike strings, they come with some degree of existing structure. This eliminates much of the parsing/unparsing overhead that is typically required when computational agents interact using string intermediate representations (parsing that, in the Unix-tools setting, is frequently done by means of heuristic, incomplete, error-prone hand-written parsers). Another Perlis aphorism makes clear the downsides of using strings as an LCD form: "The string is a stark data structure and everywhere it is passed there is much duplication of process. It is a perfect vehicle for hiding information." The problem with strings as a least-common denominator is that they are too "least," that is, too low level. We operate upon strings a character at a time—a level where it is all too easy to break the invariants of the associated grammar that typically imposes structure and meaning on the strings.

Even when s-expressions may not be the appropriate representation for the core data structures of an application, they still frequently find use around the application's "fringe," being converted to and from the internal, more highly-engineered core structures as they move across the application's boundary—with the associated benefit that it is *much* simpler and more robust to parse a tree than a string.

2 Regular trees, little languages and Scheme

Given that Lisp and Scheme programmers have been working with "semi-structured" tree data roughly three and half decades longer than XML has even existed, it is surprising that this community has never bothered to adopt one of the great, expressive tools for manipulating such data: regular trees and their associated patterns. Just as traditional regular expressions are an expressive tool for describing structure occurring within strings, regular trees can serve a similar role when dealing with recursively defined patterns occurring within trees—trees such as ones we frequently represent using Scheme s-expressions.

We have long grumbled about the lack of such tools. Each time we write a low-level Scheme macro, for example, and we find ourselves writing an incomplete and awkward syntax-checker/parser for our new form directly in Scheme (Is the form exactly four elements long? Is the second element a list of identifier/expression pairs? *Etc.*), we pause to wish for a better way. When the XML world began wisely to exploit the extensive theoretical machinery

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

developed to describe regular trees and their recognisers, we were finally pushed to carry out the design and implementation exercise we had put off so long.

The result is *trx*, a language for describing regular-tree patterns. Embedding our "little language" within Scheme provides for several benefits, which we've described elsewhere in detail [17]. On one hand, it made our task as designers and implementors easier. We only needed to design and implement tree patterns; we didn't need to implement the machinery already provided by Scheme: floating-point numbers, first-class functions, variables, loops, *etc.* On the other hand, for our programmer clients, the result was a tool that allowed regular-tree pattern matching to be tightly integrated with Scheme programs, instead of forcing this kind of operation out into a separate, distinct program written in some distinct, *ad hoc*, self-contained domain-specific language.

The rest of this paper traces out the following arc. First, we will survey the basic elements of tree automata, the underlying mathematical formalism that connects the static, declarative world of regular-tree patterns to the computational or algorithmic paradigm of their recognisers. Then we will consider the particular needs of tree pattern matching that arise when working in the setting of Scheme s-expressions. This exploration of design requirements and design rationale, plus the useful constraints imposed by the computational power of tree automata, allow us to proceed to a design for regular-tree patterns that integrates with Scheme s-expressions. The syntax and semantics of *trx* are provided in the next section, followed by examples of *trx* patterns in use. Then we will examine some details the current implementation, before concluding with a description of related and future work.

3 Overview of tree automata

Every interesting programming language is just a cover for an interesting model of computation: regular expressions and finite automata; context-free grammars and push-down automata; SQL and the relational calculus; Smalltalk and message-passing; APL and SIMD array processing; and, of course, Scheme and the λ calculus. The interesting formal model of computation underlying the design of *trx* is *finite tree automata*. The short overview that follows will spell out some of the fundamental concepts of these formal machines.

In the following sections we differentiate between *traditional* tree automata and *simplified* tree automata. This paper uses the "simplified" and "traditional" qualifiers for differentiation only; they are not part of the established nomenclature. Elsewhere in the literature, both classes of automata are referred to as tree automata interchangeably.

3.1 Traditional tree automata

Tree automata operate on labelled, finite trees: trees where every node is assigned a label f drawn from some alphabet $\mathcal{F}$. Traditional automata also require the label alphabet to be *ranked*, that is, each label has an associated natural number. Each tree node must have exactly as many children as the rank assigned its label; thus, leaf nodes are marked with rank-zero labels. We write $\mathcal{F}_n$ for the set of rank-n labels in alphabet $\mathcal{F}$.

A **traditional finite tree automaton** (FTA) over a ranked alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, where Q is a set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the form:

$$f(q_1, \ldots, q_n) \to q,$$

where $n \geq 0$, $f \in \mathcal{F}_n$, and $q, q_1, \ldots, q_n \in Q$. The symbols $q_1, \ldots, q_n$ and q are called the *initial* and *final* states of the transition, respectively.

The operation of a tree automaton involves propagating state information up (or down) through the tree. Transition rules determine how this propagation takes place. Whenever a label f is seen at a tree node and that label has the states $q_1, \ldots, q_n$ "bubbled-up" to its children, and a rule $f(q_1, \ldots, q_n) \to q$ exists in Δ, state q is propagated to the f-labelled node. In turn, the bubbled-up state then feeds into its parent node. The propagation continues until some state is bubbled up to the root node of the tree. If this state is in Q_f, then the tree term is accepted. If no final state bubbles up to the top, the tree is rejected.

In addition to the type of transition rule described above, traditional tree automata allow for ε-move transitions $q \to q'$, that occur spontaneously, changing the state assigned to a node from q to q'. Equivalence of tree automata with and without ε rules is a well-established result [6]; establishing the equivalence involves working with the ε-closure of states, *i.e.*, the set of states reachable from a state via ε-rules.

Fundamentally, every tree automaton $\mathcal{A}$ is a machine corresponding to some *tree language*. The tree language $\mathcal{L}(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all trees accepted by $\mathcal{A}$.

We've described the operation of an FTA in a bottom-up manner, but it can also be operated in a top-down manner, starting with an accept state for the root, and running the transition rules "backwards" to find the labels assigned to children, *etc.*

3.2 Simplified tree automata

A **simplified finite tree automaton** (SFTA) over an unranked alphabet $\mathcal{F}$ is a tuple $\mathcal{A}_s = (Q, \mathcal{F}, Q_i, \Delta)$, where Q is a set of states, $Q_i \subseteq Q$ is a set of initial states, and Δ is a set of transition rules. Transition rules can be either be *labelled* or *empty*:

$$f(q_{\text{in}}), q_{\text{out}} \to q \text{ or } () \to q.$$

In order to understand the way a simplified tree automaton computes, we must change our mental model of how individual nodes are "wired" together in the tree. Each node now includes a reference to its closest sibling to the right, and its leftmost child (if any, in both cases). This setup implies that at any given point in an automaton's operation, state-directed control can flow along two pathways—down to children and right to siblings. This is in contrast to traditional tree automata where state information is propagated to/from *all* children simultaneously.

A simplified automaton begins at the root of the tree, nondeterministically selecting a start state from Q_i. If the automaton is visiting an f-labelled node n while in state q, the machine selects an $f(q_{\text{in}}), q_{\text{out}} \to q$ transition. (If there is no such transition, the machine halts, reporting failure.) If n has children, the machine attempts to recursively accept them, starting in state q_{in} with n's leftmost child; if this succeeds, it then proceeds to n's siblings. If n is a leaf node, the machine checks for an empty transition $() \to q_{\text{out}}$, then proceeds to n's siblings. If the children-match attempt fails, or there is no empty rule handling the leaf node, the machine halts, reporting failure.

Figure 1
Nondeterministic simplified finite tree automaton

```
matchnode(n,q) {
  f := n.label

  /* Fail if no rule selectable. */
  Select f(q_in),q_out → q from Δ

  if n is leaf
  then matchempty(q_in)
  else matchnode(n.leftchild, q_in)

  if n has closest right sibling s
  then matchnode(s,q_out)
  else matchempty(q_out)
  }

matchempty(q) {
 if () → q ∈ Δ then return
 else fail
 }
```

To proceed to n's siblings, the machine jumps to n's closest right sibling and changes state to q_{out}. If n has no right sibling, the machine accepts iff there is an empty transition $q_{\text{out}} \rightarrow ()$. Thus empty transition rules are needed to terminate an automaton's recursive descent over a tree. Pseudocode for an SFTA is shown in figure 1.

As a trivial example, consider a regular-tree language consisting of a single term: a root node labelled with a and three child nodes labelled with b, c, d. A simplified tree automaton for recognizing such a language would have transitions

$$\begin{array}{ll} a(q_1),q_2 \rightarrow q_0 & b(q_2),q_4 \rightarrow q_1 \\ c(q_2),q_6 \rightarrow q_4 & d(q_2),q_2 \rightarrow q_6 \\ () \rightarrow q_2 & \end{array}$$

with $Q_i = \{q_0\}$.

Note that the label alphabet $\mathcal{F}$ is unranked in the sense that when a node labelled f is processed, the automaton is only concerned with the presence of the node's leftmost child and closest right sibling. Nothing in the rule format enforces how many children or siblings a given node is allowed to have. A simplified automaton can fix the number of children permitted a tree node by encoding this in the states traversed as it scans across the siblings, but it may also permit a child to have an arbitrary number of children, a degree of power not available with traditional automata. Thus simplified automata are strictly more powerful than traditional automata. This power is useful for the kinds of s-expression and XML trees we process in the real world.

3.3 Converting between models

A traditional tree automaton can be converted to an equivalent simplified tree automaton in the following way. Starting from each state q of the initial states, the algorithm selects all the rules that have q as their final state. For each rule $f(q_1,\ldots,q_n) \rightarrow q$, a labelled rule $f(q_1),q' \rightarrow q$ and an empty rule $() \rightarrow q'$ are added to the simplified automaton, for fresh state q'. Then a new state q'' and empty rule $() \rightarrow q''$ are added to the automaton, for fresh state q''. The children states are processed similarly, except that instead of generating a fresh "out" state as we did for the rules starting in q, the rules starting in q_n are processed recursively with their "out" state as their right sibling's final state q_{n+1}. The "out" state for the rightmost sibling is the newly-generated q''.

A simplified tree automaton resulting from the conversion recognizes the same language as its traditional equivalent. However, the arity information is now encoded directly in the rules; symbols no longer have an intrinsic arity attached to them.

A conversion of a simplified tree automaton to an equivalent traditional tree automaton is not possible in the general case: as we've seen, traditional tree automata cannot handle symbols with unbounded arity, and thus are strictly less powerful.

3.4 Nondeterminism in tree automata

Both of the above definitions describe non-deterministic automata, *i.e.*, automata that can "fork" in a number of directions if multiple transitions can be applied in a given machine configuration. As with regular-string languages, a deterministic variant of tree automata can be defined as a subset of the general nondeterministic one.

The equivalence of deterministic and non-deterministic automata has been established for traditional tree automata [6]. A traditional tree automaton is said to be **deterministic** (DFTA) if there are no rules with the same left-hand side, and no ε-rules.

The construction of an equivalent automaton proceeds as follows. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a non-deterministic tree automaton. Then there is an equivalent DFTA, $\mathcal{A}_d = (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$ such that (1) every state in Q_d is a non-empty set of original states in Q, and (2) every rule in Δ_d is computed with

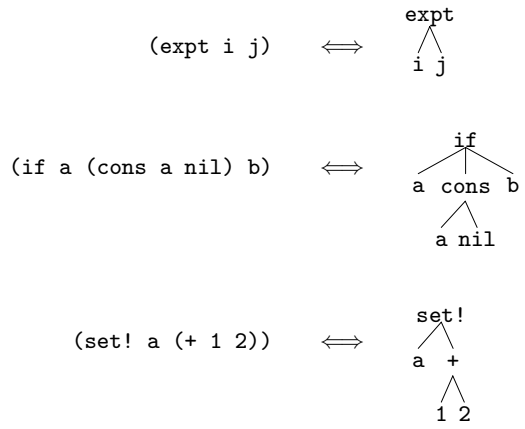
$$\begin{array}{l} f(s_1,\ldots,s_n) \rightarrow s' \in \Delta_d \text{ iff} \\ s = \{q \mid q_1 \in s_1,\ldots,q_n \in s_n, f(q_1,\ldots,q_n) \rightarrow q \in \Delta\} \\ s' = \varepsilon\text{-closure}(s) \end{array}$$

Just as with regular-string expressions and their associated finite-state automata, this power-set construction of an equivalent deterministic tree automaton may, in the worst case, result in exponential state explosion [6]. And just as with regular-string expressions, this negative effect is offset by the fact that deterministic tree automata are considerably faster at parsing certain regular-tree languages [6], in the standard space *vs.* search trade-off.

We are not aware of any results pertaining to equivalence of deterministic and non-deterministic simplified tree automata. Our SFTA technology works strictly with non-deterministic machines—that is, it manages non-determinacy at run time by performing backtracking search.

4 S-expressions, XML, and regular trees

Lisp s-expressions are frequently used to represent labelled trees, using the encoding that an internal tree node is represented by a list whose head is its label and whose tail is the list of its children; leaf nodes are simply represented by non-list data. The following schematics illustrate the mapping:



In the domain of XML, the delimiting characters `(` and `)` are replaced with <*tag attr-list*> and </*tag*>, with *tag* serving as the node's label. This treatment slightly oversimplifies the way XML documents are put together, but it exposes the features common to labelled-tree languages at large, which is what the *trx* language is intended to process. The XML community has developed a plethora of language tools for describing regular-tree patterns as well as transducers operating on the trees matched by these patterns [12, 4, 6, 21, 3]. Although these tools are outside the scope of this paper, their existence reaffirms the utility of convenient, robust tools for regular-tree processing.

4.1 Variable-arity constructors

One issue that arises when considering tree structure in XML and symbol-labelled s-expression trees is the possibility of variable-arity constructors (that is, variable-rank tree labels). Both `(+ 3 7)` and `(+ 2 6 3 1)` are legal Scheme expressions, yet the `+` label must be assigned a fixed arity in order for a traditional tree-automata to be able to match it. These cases arise in both XML and s-expression trees, and when they do, we must resort to the more powerful model of simplified tree automata.

4.2 Unlabelled tree nodes

Another issue we find in the context of interpreting s-expressions as labelled trees is that in some s-expressions, not all tree nodes are labelled. For example, consider the structure of a Scheme `let` form, which has the following syntax:

`(let ((`*var exp*`) ...)` *body* `...)`

The first child of a `let` node, the bindings list `((`*var exp*`) ...)`, has no label. (As we discussed above, it is also variable-arity.) We can find the same problem in the syntax of the individual clauses of a `cond` expression, or even in the "default" syntax of Scheme function calls, which are not introduced by any kind of `call` keyword.

It is not technically difficult to handle unlabelled nodes within the finite-automaton model; our implementation does so by introducing special anonymous symbols that have unique names with respect to the rest of the automaton's label set. The critical point is that, to handle this tree idiom, which occurs in common practice with s-expressions, we must account for them in the design of our pattern notation.

4.3 Factoring pattern, automaton and data

One theme in the design of *trx* is factoring the layers of the design. Whether the terms under consideration are s-expressions, XML documents, or some other form of tree data, the basic pattern notation and the underlying abstract automata models, which specify the basic processing engine for tree terms, should remain unaltered. We'll return to this factoring in the discussion of the implementation.

Another design concern was abstracting over the nuts and bolts of finite-tree automata or other possible semantic engines for *trx*. (*E.g.*, is a pattern implemented as a non-deterministic traditional FTA using search, as a deterministic traditional FTA without backtracking, or with an SFTA?) Where possible, we kept the notation and its semantics independent of these implementation pragmatics (adding variable-arity patterns in the pattern notation does restrict the space of possible implementations, however, so this is not 100% possible).

4.4 Escapes to Scheme

We've found it useful in previous little-language designs to provide mechanisms not only for embedding the little language within Scheme, but for embedding general Scheme within the little language. This, of course, dramatically changes the power of the pattern model, allowing us to define pattern matchers whose top-level control skeleton is an FTA, but who may invoke arbitrary computation at the "leaves" of the computation. Adding such a facility has an impact on the implementation of the system, restricting our ability to statically analyse patterns (a price we pay for the increased computational power), and requiring the implementation to be written so it can simply pass the embedded Scheme code through the pattern compiler, to be dropped into place in the final output.

As we'll see, the ability to invoke arbitrary Scheme at the leaves of the pattern matcher in particular allow us to have trees whose leaf nodes are not just symbols, but any kind of data. This is important in the world of Scheme s-expressions, which are frequently composed of more than symbols and parentheses—they may, for example, contain records or booleans or strings. We might, in some contexts, wish to permit only leaves that are positive integers between 0 and 100—something which does not fit the basic tree-automata model, which discriminates only on the symbols that label nodes, including leaf nodes.

Similarly, when a user writes down a regular-tree expression to describe the syntax of the Scheme `let` form, he will want to capture in the pattern the fact that the left-hand sides of the binding forms can be any symbol at all... but only a symbol, not a general subtree.[1] Allowing escapes to general Scheme code permits us to write

[1] If he wants to capture the constraint that these bound identifiers must be distinct from one another, he's completely outside the power of the regular-tree model. The price we pay for specialised notations and restricted computational models is that we can't solve all possible problems. Note that our hypothetical programmer could always use a more complex escape to Scheme to check this distinct-identifier constraint, or defer it to a later check. This is analogous to the way compilers detect some illegal programs while parsing (*i.e.*, syntax errors), leaving others for later static analysis to find (*e.g.*, type errors). This distinction happens for the same reason—the expressiveness of context-free grammars and the power of the associated push-down automata that recognise their languages are restricted, making them unable to encode all the static constraints

such patterns, yet remain in the tree-automaton/declarative-pattern model for the most part. This functionality is conceptually aligned with the Scheme's own type system where types are typically defined by means of general Scheme predicates which discriminate between members and non-members of the type, *e.g.*, functions such as `list?`, `string?`, `symbol?`, and so forth.

4.5 Dynamic patterns

Besides inlining Scheme predicates to match tree leaves, we might also want to escape to Scheme code within a pattern in order to *compute* a sub-pattern. This allows users to dynamically stitch tree automata together, or construct patterns which may have a run-time dependency on particular computations or input data.

4.6 Collecting submatch data

We frequently want our patterns to do more than simply recognise trees, reporting only a "yes" or a "no." In many cases, we want to use our patterns to *select* indicated parts of a tree. One mechanism for doing so is to add elements to the pattern language for marking components of a tree that match particular pieces of a pattern. On a successful match, the selected sub-trees are then returned to the programmer as a result. String regular expressions frequently have similar kinds of support for picking out elements of a matched string. Providing submatches in the pattern notation complicates the implementation of the pattern matcher; in particular, the pattern optimiser has to be careful not to optimize away subpatterns that contain a submatch.

5 The *trx* language

The syntax of *trx* patterns takes the form of the familiar s-expression and borrows extensively from the *SRE* regular-expression notation introduced in *scsh* [19, 18]. The grammar is given in figure 2.

A regular-tree expression (or pattern) denotes a set of trees. A pattern which is simply a literal, such as the number 5, is a pattern matching only the leaf tree 5. Similarly, the pattern `'fred` (or, equivalently, `(quote fred)`) matches the leaf which is the symbol `fred`.

The pattern `(@` *symbol rte* `...)` matches a tree whose root is labelled with *symbol*, and whose children match the *rte* sub-patterns. When *symbol* doesn't conflict with one of the pattern keywords, the `@` can be elided. A tree with an unlabelled root can be matched with a `(^` *rte* `...)` pattern.

We introduce choice with the pattern `(|` *rte* `...)` which matches any tree matched by any of the *rte* subforms.

The pattern `(any)` matches any tree. We can write a match which matches no tree with the empty-choice pattern `(|)`. This is not particularly useful for user-written patterns, but could be useful for patterns produced mechanically, either from higher-level macros or dynamically in response to program input.

The sequence operators `*`, `+` and `?` match zero-or-more, one-or-more and zero-or-one trees matching their subpattern, respectively.

of a well-formed legal program. The role of a little language is to make the common cases easy; the role of a general purpose language (such as our escapes to Scheme) is to make the rest of the cases possible.

What makes regular-tree patterns interesting is recursion in the patterns. This is introduced with the `rec` and `letrec` forms. The pattern `(rec` *ident rte*`)` matches a tree that matches *rte*, with the proviso that free references to *ident* in *rte* must recursively match the pattern, as well. Thus we can describe a pattern that matches binary trees whose internal nodes are labelled `+` and whose leaves are 42 with the pattern

```
(rec t (| 42 (@ + t t)))
```

This would match any of the trees `42`, `(+ 42 42)`, `(+ 42 (+ 42 42))`, `(+ (+ 42 42) 42)` and so forth.

The `letrec` form allows mutual recursion by binding the pattern identifiers in a recursive scope. We can also bind pattern identifiers with simple lexical scope with the `let` form.

The `(submatch` *rte*`)` form lets us mark a part of a larger pattern to indicate to the pattern matcher that, in the event of a complete match, the sub-trees matching *rte* should be retained for later retrieval. Note that a single submatch can match more than a single tree term. For instance, the patterns

```
(rec t (| 42 (@ + (submatch t) t)))
(@ + (* (submatch 42)))
```

would produce, upon a successful match, a variable number of submatches depending on the height and width of the tree term. The matcher produces a list of terms for any single `submatch` form, ordered according to the pre-order position of submatched terms in the original tree. Thus for pattern

```
(rec t (| ,number? (submatch (@ + t t))))
```

and tree term `(+ (+ 1 2) (+ 3 4))`, the list of saved items for the submatch will consist of every internal node in the source tree:

```
((+ (+ 1 2) (+ 3 4))
 (+ 1 2)
 (+ 3 4))
```

Finally, we can escape to general Scheme code in two different ways. The pattern `,`*exp* allows us to write a Scheme expression providing a general predicate which accepts or rejects trees. Thus we can change our sum-of-42s example above to be a sum tree for general numbers with the pattern

```
(rec t (| ,number? (@ + t t)))
```

or a sum tree of even numbers with the pattern

```
(rec t (| ,(λ (x) (and (number? x)
                       (even? x)))
          (@ + t t)))
```

The pattern `,@`*scheme-exp*, allows us to write a Scheme expression that itself evaluates to a *trx* pattern value, which is then plugged into the enclosing pattern. This allows us to dynamically construct *trx* patterns, instead of restricting them to patterns that are completely fixed at compile time. (Consequently, this feature has major implications on the compile-time handling of patterns—when it is used, we must do a kind of "partial evaluation" of the pattern, deferring the processing of dynamic components to run time. Fortunately, we *can* statically determine if a particular pattern uses this feature of the language, and so only need defer such processing with patterns that do so. So the extra overhead of dynamic pattern construction is only invoked as needed, making it a pay-as-you-go feature.)

Figure 2 Syntax of trx regular-tree expressions.

```
rte ::= literal | 'symbol                      ;Literal atom
      | (@ symbol rte ...)                     ;Tree with root labelled symbol
      | (symbol rte ...)                       ;As for @, when no ambiguity.
      | (^ rte ...)                            ;Tree with unlabelled root
      | (any)                                  ;Matches any tree
      | (| rte ...)                            ;Choice
      | (* rte)                                ;Matches a sequence of [0,∞) rte's
      | (+ rte)                                ;Matches a sequence of [1,∞) rte's
      | (? rte)                                ;Matches a sequence of [0,1] rte's
      | (rec ident rte)                        ;Recursively defined pattern
      | (let  ((ident rte) ...)  rte)          ;Lexical pattern binding
      | (let* ((ident rte) ...)  rte)          ;Lexical pattern binding
      | (letrec ((ident rte) ...)  rte)        ;Pattern with mutual recursion
      | ident                                  ;Reference to pattern bound by rec, let or letrec
      | (submatch rte)                         ;Matched subtree saved for subsequent retrieval
      | ,scheme-exp                            ;General predicate
      | ,@scheme-exp                           ;Dynamically computed tree automaton

ident ::= symbol
literal ::= number | string | boolean | char
```

As an example putting multiple components of the language together, a pattern which specifies the syntax of the Scheme `let` expression is

```
(@ let (^ (* (^ ,symbol? (any))))
   (+ (any)))
```

or, with components of the pattern let-bound for clarity,

```
(let* ((binding  (^ ,symbol? (any)))
       (bindings (^ (* binding)))
       (body     (+ (any))))
  (@ let bindings body))
```

Notice how the unlabelled-tree patterns are used to match each (*var exp*) binding form as well as the list of these bindings. As an exercise, you may wish to extend the pattern to handle named-let forms used for iteration.

6 Static semantics

Our informal description of the *trx* language has glossed over a distinction between the language's various constructs. While an operator such as `@` produces a pattern that matches a tree, the `*`, `+` and `?` operators produce patterns that match a *sequence* of trees. These sequence or "forest" patterns can appear anywhere in the language a tree pattern can appear, with the restriction that a complete, top-level pattern cannot be a sequence pattern. We cannot encode this directly in the grammar due to the presence of the pattern-binding forms (`let`, `letrec` and `rec`)—there's no way to design the grammar to guarantee that a particular identifier reference is made to a tree-pattern binding and not a forest-pattern binding. This is the sort of restriction that one typically manages in the post-parse static-semantics phase of a compiler, in a type-system-like manner. This is exactly what we do. The macros that process tree patterns check them to ensure that the top-level pattern has a "tree-pattern" type. Similarly, because identifier references are resolved lexically, references to unbound identifiers are checked for and rejected at macro-expansion time.

7 Examples

At last, we present a set of examples which work to illustrate the capabilities of the *trx* language. We embed patterns into Scheme code by means of the Scheme form (`trx` *rte*). This is a Scheme expression whose body *rte* is not Scheme code, but rather a *trx* regular-tree pattern. The `trx` form produces a tree-automaton value which can be passed to the `trx-match` pattern matcher. It is implemented as a macro that compiles its pattern body to a tree automaton, represented with an abstract data type. More details of the implementation are given in the following section. The matcher function is invoked as (`trx-match` *pat s-exp*), taking a tree automaton *pat* (produced by a (`trx` *rte*) form), and a tree *s-exp* to which it should be applied. It returns a non-false value for a successful match, and `#f` otherwise.

Example 1 We begin with a set of Scheme expressions that construct nested lists of numbers. The number leaves are matched using the Scheme `number?` procedure. The example demonstrates escaped Scheme code and the use of the `rec` operator.

```
(let ((p (trx (rec q (| (cons q q)
                        (cons ,number? q)
                        nil)))))
  (trx-match p '(cons 1 nil))      ; match
  (trx-match p '(cons nil nil))    ; match
  (trx-match p '(cons (cons a nil) ; fail
                      (cons 1 nil))))
```

Example 2 A somewhat more interesting use of escapes to Scheme code involves user input. The language is similar to the first example, except that numbers must be divisible by the user-specified divisor.

```
(let* ((i (read))
       (idiv? (λ (n) (= (modulo n i) 0)))
       (p (trx (rec q (| (cons q q)
                         (cons ,idiv? q)
                         nil)))))
  (trx-match p '(cons nil nil))      ; match
  (trx-match p '(cons (cons a nil)   ; fail
                      (cons 1 nil))))
```

Example 3 The purpose of this example is to illustrate the use of the `letrec` construct. The pattern below matches any Scheme expression that consists solely of applications of + and * such that + is never applied to a + expression, and *vice versa*. In other words, the constructors must alternate within the tree. This example also illustrates the use of variable-arity constructors and labels that collide with reserved keywords.

```
(let ((p (trx (letrec ((m (| n (@ * (* a))))
                       (a (| n (@ + (* m))))
                       (n ,number?))
                (| m a)))))
   (trx-match p '(* 2 (+ 3 4))) ; match
   (trx-match p '(* 2 (* 3 4)))); fail
```

Example 4 We now consider a pattern for recognizing XML-like data sets capturing data entered into an online purchase-order form.

```
(trx (order (date ,string?)
            (shipto (name ,string?)
                    (address ,string?))
            (? (comment ,string?))
            (+ (item (part ,number?)
                     (quantity ,number?)
                     (price ,number?)))))
```

The pattern will match

```
(order (date "2004-06-11")
       (shipto (name "Bill")
               (address "1 Main Street"))
       (comment "Please, hurry!")
       (item (part 111)
             (quantity 1)
             (price 1.00))
       (item (part 222)
             (quantity 2)
             (price 2.00)))
```

but not

```
(order (date "2004-06-11")
       (comment "Please, hurry!")
       (shipto (name "Bill")
               (address "1 Main Street"))
       (item (part 111)
             (quantity 1)
             (price 1.00))
       (item (part 222)
             (quantity 2)
             (price 2.00)))
```

or

```
(order (date "2004-06-11")
       (shipto (name "Bill")
               (address "1 Main Street"))
       (comment "Please, hurry!"))
```

Example 5 Building on the previous example, we illustrate how the `submatch` operator can be used to transduce tree terms. Specifically, we want to match a sequence of orders, but we also want to add a free gift from the company to every order in the sequence which includes two or more distinct items. We augment the previous definition of an order-matching pattern to match "sequence-of-orders" datasets. Figure 3 shows how we can match a suitable sequence of orders against this new pattern, alter submatched terms, and reconstitute them into a new list of orders.

Example 6 As a final example of *trx*, figure 4 shows the grammar of *trx*, expressed as a *trx* pattern. It's not an accident that this is so straightforward to encode: the long-standing convention s-expression language designers use for their syntax design[2] is *exactly* the labelled-tree model processed by tree automata. So this example gives away one of the elements of our development agenda. We are interested in the use of *trx* to provide more precise syntax specification and error-checking for the kinds of languages we like: the s-expression-based ones.

8 Implementation

We have implemented the *trx* system as a module in the scsh Scheme environment [19]. The code is fairly portable; its most significant element of non-portability is its use of a non-R5RS low-level macro system. Our implementation can be subdivided into following components:

- A macro embedding applications of the *trx* notation into Scheme forms. Basically, the macro simply interfaces the *trx* compiler to the underlying Scheme compiler.
- A set of Scheme data-type definitions encoding the abstract syntax of the *trx* language as well as the resulting automata values. This establishes a set of ADTs around which processing of regular-tree patterns takes place.
- A pair of Scheme procedures for parsing tree patterns from their s-expression concrete syntax into their representation using the internal AST structures; and for unparsing from an AST value back to its external, printable s-expression representation.
- A procedure that translates a *trx* AST into an automata value. This is the heart of the *trx* compiler.
- Pattern-matching procedures. One of these procedures is the pattern matcher; the other is a routine that extracts submatched terms from a result of a successful match.

8.1 The *trx* macro

As mentioned above, the *trx* compiler is invoked on every occurrence of the `trx` macro in Scheme source code. In our implementation, we utilize Scheme 48's low-level macro facility, the Clinger/Rees "explicit renaming" macros [5], which allows both full control of hygiene and permits macros to be written in general Scheme code. The latter feature is particularly important, as the *trx* machinery is fairly complex—at the complexity level of a small compiler, as opposed to the kind of simple pattern-directed extensions usually implemented via the R5RS high-level macro facility. The macro simply invokes the rest of the machinery, which parses the pattern into an AST, performs static-semantics checks, simplifies the pattern, compiles it to a value in the automata ADT, and finally renders the result automaton as a block of Scheme code.

[2] Barring certain regrettable exercises in syntactic excess, such as the Common Lisp `loop` form.

Figure 3 A simple tree transducer.

```
(let ((pat (trx (order (date ,string?)
                       (shipto (name ,string?)
                               (address ,string?))
                       (? (comment ,string?))
                       (submatch (+ (item (part ,number?)
                                          (quantity ,number?)
                                          (price ,number?)))))))
  (map (λ (order) (cond ((trx-match pat order) =>
                         (λ (match) (let ((items (trx-submatch match 1)))
                                      (if (>= (length items) 2)
                                          (cons '(item (part 001)
                                                       (quantity 1)
                                                       (price 0))
                                                items)
                                          items))))
                        (else (error "Illegal order"))))
       orders))
```

Figure 4 The *trx* grammar as a *trx* pattern.

```
(rec rte (| ,string? ,number? ,character?           ; Literals
            ,(λ (x) (or (not x) (eq? x #t)))        ; boolean?
            (@ quote ,symbol?)                      ; 'symbol
            (@ @ (* rte))                           ; (@ rte ...)
            (^ ,(λ (x)                              ; (symbol rte ...)
                  (and (symbol? x)
                       (not (member? x '(quote @ ^ any or * + ?
                                         rec let let* letrec submatch
                                         unquote splicing-unquote)))))
               (* rte))
            (@ ^ (* rte))                            ; (^ rte ...)
            (@ any)                                  ; (any)
            (@ | (* rte))                            ; (| rte ...)
            (@ * rte)                                ; (* rte)
            (@ + rte)                                ; (+ rte)
            (@ ? rte)                                ; (? rte)
            (@ rec ,symbol? rte)                     ; (rec id rte)
            (@ let    (^ (* (^ ,symbol? rte))) rte) ; (let    ((id rte) ...) rte)
            (@ letrec (^ (* (^ ,symbol? rte))) rte) ; (letrec ((id rte) ...) rte)
            (@ submatch rte)                         ; (submatch rte)
            (@ unquote (any))                        ; ,scheme-exp
            (@ unquote-splicing (any))))             ; ,@scheme-exp
```

If the pattern contains dynamically-computed components, then those parts of the pattern are not known at compile time. In these cases, the macro expands into Scheme code that is essentially a template for the AST—code which will compute the dynamic components and then assemble the rest of the AST around them. This AST-assembly code is then inserted into code which will invoke the pattern compiler on the AST. Essentially, the macro arranges for the compiler invocation it represents to be delayed to run time, thus deferring production of the final automaton to run time. With this exception of handling dynamic components, compiling a *trx* pattern happens entirely at macro-expansion time (that is, compile time). By run time, a source *trx* pattern has already been converted to an equivalent automaton value.

8.2 Abstract syntax

Abstract syntax consists of a handful of record definitions that encode nodes of a *trx* abstract syntax tree (AST). Employing an AST allows us to make the *trx* tool chain independent of the details of our concrete notation; one could try out alternate syntaxes without much work. Furthermore, processing that can be done on the AST can be shared by distinct back-ends that might target different automata models or implementations of those automata.

The AST is defined using a set of record types. Some examples are

```
(define-record ast-sym-node
  symbol    ; Label of root
  children  ; Child patterns
  private)

(define-record ast-seq-node
  quantifier ; One of * + ?
  child      ; Child node
  private)

(define-record ast-choice-node ; (| ...) node
  children   ; List of nodes
  private)

(define-record ast-code-node  ; ,<scheme> node
  code       ; The <scheme> exp (as an s-exp)
  private)
```

Note that each of the records contains a special `private` field. This field is used by the compiler to manage accounting information. For instance, the `private` field tells us, among other things, whether a node has already been visited by the compiler (which is not uncommon due to prevalence of cycles).

Another interesting datatype that is there purely for the convenience of the compiler developers is the AST "handle" node.

```
(define-record ast-handle
  ref)  ; a reference to actual AST
```

Handle nodes are useful when translating recursive patterns, *i.e.*, patterns that begin with `rec` and `letrec`. When these patterns are compiled it is common for one part of the pattern to refer to another part of the pattern that has not been compiled yet. We address this problem by referencing all recursive patterns through a handle node which is first created without a reference field set and is later filled with the reference to the actual abstract syntax tree.

8.3 Automata values

Once an AST is constructed, it must be converted to an automata value. An automata value is represented with yet another record datatype.

```
(define-record sfta
  states             ; Symbol list
  alphabet           ; Symbol list
  labeled-rules
  empty-rules
  final-states)
```

The labelled rules are encoded with

```
(define-record label-rule
  sym-name
  in-state
  out-state
  final-state)
```

and empty rules with

```
(define-record empty-rule
  final-state)
```

Note that the `sfta` record doesn't have fields to support the full requirements of the *trx* notation, such as dynamically-created automata, escapes to Scheme code, and submatches. We delegate tracking of this information to another record datatype:

```
(define-record complex-fta
  sfta            ; Finite tree automaton
  special-states  ; State->inlined-code alist
  submatch-states); Submatched states
```

The `special-states` field is an association list of states and suspended lambda values that correspond to individuals chunks of Scheme code inlined in the *trx* notation. The `submatch-states` field is a list of states at which submatches are to be saved for later retrieval.

Note that the simplified automata ADT permits multiple backends for different models of execution. The one we have implemented uses the automaton itself as a run-time value which is passed, along with a subject tree, to a backtracking SFTA interpreter for execution, which proceeds in a top-down manner. One could consider, alternatively, compiling an SFTA directly to Scheme code; this is something we would like to do.

An earlier version of the system had support for both traditional and simplified automata. The choice in the type of automata value was guided by whether the source pattern included variable-arity constructors. If it did, a simplified automaton would be produced; otherwise, a traditional automaton would be produced by default. Keeping traditional automata in the system allowed for the possibility of "compiling away search" by expanding a non-deterministic TFTA to a deterministic one, buying execution speed for the price of extra compile time and potential state-space explosion. The implementation left the choice of whether to search with a small non-deterministic machine or do fast, non-backtracking execution with an expanded deterministic one in the hands of the application programmer.

We subsequently dropped traditional automata as one of the alternative backend models of computation. (The factoring of the au-

tomata ADT into the `sfta` and `complex-fta` records is, in fact, a relic of this earlier implementation—both traditional and simplified FTAs shared inlined-code and submatch annotations by means of the common `complex-fta` record.) As mentioned in Section 3.2, traditional automata are strictly less powerful than simplified tree automata. Keeping two backends to the *trx* compiler did nothing to expand the semantic power of the notation, while it did considerably increase the complexity of our code. Simplifying the implementation made it easier for us to focus on the design of the language; we may revisit automata determinisation as an implementation technology at a later date.

8.4 Compilation

Thus far we have described the way the *trx* compiler is invoked and the types of intermediate and final values it generates. We now describe how these AST and automata ADT values are generated.

The source-level concrete-syntax pattern s-expression is parsed into an AST with a simple recursive translation; the static semantics of the AST are likewise checked with a simple recursive tree-walk that "type checks" the identifier bindings and references.

Translation between ASTs and automata values is a bit more complex. We provide a high-level description of the algorithm, which generates a set of labelled and empty rules for an SFTA from a given abstract-syntax tree.

In the case of a `ast-sym-node`, this entails generating a fresh state that is the "out" state for the label, and a fresh state that is the "out" state for the rightmost child of the AST node. We add empty transitions leading to both of these states. We then obtain the label's "in" state by folding states, beginning with the rightmost child's "out" state, across the children nodes processing each one of the children recursively. Given a label's "out" and "in" states, we generate a fresh final state and add the corresponding rule for that label.

When processing `ast-seq-node` nodes, we restrict the compiler to generate only rules with the same "out" and "final" state. This restriction is necessary to capture the fact that for patterns of the form `(*` *pattern*`)`, the final state after parsing one term matching *pattern* is the same as the "out" state used in translating the term's left sibling. Translation of other AST datatypes is straightforward and follows the same general template.

Macros must produce concrete s-expressions—that is, Scheme source to be handed to the Scheme compiler. Automata values, which are defined as records, do not qualify as such. As a final step, then, the compiler translates the automaton, represented with the ADT, into a Scheme expression describing the direct construction of that value, as a tree of calls to the record constructors. So the `trx` macro finally expands into Scheme code that, when executed at run time, will construct the automaton (as an ADT) used to match the pattern.

8.5 AST-to-AST optimizations

In addition to pattern-to-automaton translation, our compiler also performs some simple optimizations on pattern ASTs. These optimizations are beneficial because the reduced ASTs result in smaller equivalent automata. Some example optimizations are:

- **Propagating `(any)` matches**
  If an `(any)` match is encountered in one of the arms of a choice clause, then the whole clause may be replaced with an `(any)` node. This simplification can bubble up the tree.
- **Propagating dead matches**
  A dead match `(|)` usually allows its containing form to be reduced to a dead match, as well. This simplification also bubbles up through the tree.
- **Merging choice nodes**
  If `(| ...)` forms are nested, they can be flattened into a single such form.

The presence of submatch forms in deleted code can allow an observer to detect some of the transformations, so care must be taken in these cases not to simplify away `submatch` forms that might bind data in the original pattern.

8.6 Executing automata

Our implementation provides a single pattern matcher, the procedure `trx-match`, that takes an tree automaton and an s-expression. The result of a successful match is a *match record* that contains for every submatch (in order of occurrence of the `submatch` clause in the original pattern) a list of submatched terms. We provide a special procedure `(trx-submatch` *m i*`)` for retrieval of submatched information, where *m* is the match record, and *i* is the index of a particular submatch form in the original pattern. Submatch forms are assigned a match-record index in the top/down, left-to-right preorder of the pattern.

The implementation of `trx-match` is a simple SFTA interpreter, which is a pretty direct transcription of the pseudocode of figure 1 into Scheme.

9 Related work

Trx closely follows the choices made in the design of *rx* low-level macro and its associated *sre* regular-expression forms, originally conceived for the *scsh* environment [18]. Similar high-level semantic features, such as choice, repetition, submatches, and the inlining of Scheme code, have similar syntactic encodings in both languages. We were thus able to leverage the design work that went into the *sre* system, and wel also hope that this will make it easier for programmers familiar with the *sre* notation to read and write *trx* patterns, mapping intuition gained from dealing with string-matching patterns to problems in the completely different domain of trees and their patterns.

Tree automata, which provide the foundation of *trx*'s semantics, have enjoyed a consistent flow of research contributions over the last three decades [20, 10, 8, 9]. A resurgence of interest in the late 90s coincided with emergence of semi-structured and tree-structured data, especially in reference to XML [1, 12, 2]. Incidentally, the programming languages built to describe and manipulate tree-structured data have been consistently targeted at handling XML [11, 7]. *trx* differentiates itself from these efforts by combining the benefits of a domain-specific language with the benefits of being able to leverage the functionality of the host language. The authors are not aware of another regular-tree pattern language that is not a standalone domain-specific language.

While we have chosen to embed our little language within Scheme due to the ease of inventing new constructs and translating them into the host language, the functional-programming paradigm is equally valuable to the parsing algorithm, its utility already recognized in

the context of XML validation [13]. As evident from the description of the way tree automata process their input, the subsequent application of automata rules to the top-level term and its subterms is naturally recursive.

10 Future work

We are currently developing an SFTA-to-Scheme compiler that will allow static *trx* patterns to be expanded directly into executable Scheme code, rather than requiring an SFTA interpreter. Although there is an existing SFTA-to-Scheme compiler [15], it operates on a more restrictive language than *trx*. For instance, that language does not handle patterns of the form `(| (@` l_1 `...) (@` l_2 `...))` when the labels l_1 and l_2 are identical. We are currently investigating ways to overcome this limitation.

One clear limitation of *trx* is that it builds upon but a tiny fraction of the research work available in the domain of tree automata, semi-structured and tree-structured data, and XML processing. In the future we would like to mine the designs of the existing Scheme-based XML-processing tools SXML and SXSLT [14] to enhance the feature set of *trx*. These systems already employ the functional-programming paradigm to manipulate XML data and have a facility for encoding XML data as s-expressions. Both of these features are consistent with the design goals for *trx*.

We also wish to extend *trx* to incorporate functionality such as ML-style pattern matching and richer functionality for tree matching and transformation. For example, it would be very useful for certain classes of tree structures (such as the mail-order structures given in example 4), to have a pattern that matches, not a sequence of child patterns occuring in a fixed order, but rather the set of child patterns allowed to occur in any order. For example, we could specify that an *order* node must have a *date*, *shipto*, optional *comment*, and *item* child node—but that these children may occur in any order. This would provide logarithmic compression of unordered-sequence patterns, greatly simplifying these kinds of patterns. Some XML pattern-matchers provide this kind of functionality; we'd like to add it to the *trx* notation.

Finally, application of the notation to help write real programs will provide the most valuable feedback on the design of the language, exposing shortcomings and potential areas of extension. (In fact, we're already dissatisfied with the submatch facility and intend to redesign it.) We look forward to gaining more experience with uses of *trx*.

11 Conclusions

Now that the rest of the world has caught on to the benefits of working with semi-structured data with a fixed concrete representation, the importance of tools that help operate on this kind of data is only going to increase—it's reasonable to assume that in the very near future, a significant percentage of the world's data is going to be stored in XML format. The *trx* pattern language, or some future revision of it, can help Scheme programmers describe and operate on this data. Note that while our implementation of *trx* provides for matching patterns against trees that come in the form of s-expressions, neither the design nor the implementation is s-expression specific. Using the notation and adapting our implementation to allow matching and transforming other kinds of labelled trees—such as XML—would not be difficult. Almost the entire code base could be reused in a modular way. The implementor would only need to write a new SFTA interpreter (or compiler) that allows SFTAs to operate upon the new tree structures.

Note also that, just as the Lisp community stole a 40-year march on the rest of the world by adopting semi-structured data early, we have other technologies that continue to bring advantage to the Scheme-programming experience. Chief among these is the ability of Scheme programmers to tightly integrate little languages within Scheme by means of the powerful Scheme macro system. This means that (1) domain-specific extensions can focus on their domain-specific components without needing to re-invent the entire wheel of a general-purpose programming language, and (2) different components of a system written with different domain-specific extensions can be closely coupled within the same program, rather than needing to appear in two completely distinct programs written in two completely distinct domain-specific languages.

This is exactly the story of *trx*—exploiting the domain-specific expressiveness of regular-tree patterns within the powerful, general-purpose framework of the Scheme language.

12 References

[1] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML). *The World Wide Web Journal*, 2(4):29–66, 1997.

[2] F. Bry and S. Schaffert. Pattern queries for xml and semistructured data. Technical report, Institute for Computer Sciences, University of Munich, 2002.

[3] R. D. Cameron. Rex: Xml shallow parsing with regular expressions. Technical report, Simon Fraser University, 1998.

[4] B. Chidlovskii. Using regular tree automata as xml schemas. In *Proc. IEEE Advances in Digital Libraries*, pages 89–104, 2000.

[5] W. Clinger and J. Rees. Macros that work. In *Proceedings of Conference on Principles of Programming Languages*, pages 155–162, 1991.

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. released October, 1 2002.

[7] V. Gapeyev and B. Pierce. Regular object types, 2003.

[8] F. G. Gécseg and M. Steinby. *Tree automata*. Akademiai Kiado, 1984.

[9] F. G. Gécseg and M. Steinby. *Handbook of Formal languages*, volume 3, chapter Tree languages. Springer-Verlag, 1997.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[11] H. Hosoya and B. C. Pierce. "XDuce: A typed XML processing language". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[12] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[13] O. Kiselyov. A better XML parser through functional programming. *Lecture Notes in Computer Science*, 2257, 2001.

[14] O. Kiselyov and K. Lisovsky. XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT. In *International Lisp Conference*, New York, NY, 2003.

[15] M. Y. Levin. Compiling regular patterns. In *Proceedings of*

the eighth ACM SIGPLAN international conference on Functional programming, pages 65–77. ACM Press, 2003.

[16] Alan J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9), September 1982.

[17] O. Shivers. A universal scripting framework or lambda: the ultimate "little language". In *Proceedings of Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference, ASIAN*, volume 1179 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 1996.

[18] O. Shivers, B. D. Carlstrom, M. Gasbichler, and M. Sperber. *Scsh Reference Manual*, 0.6.4 edition, April 2003.

[19] Olin Shivers. A scheme shell. *Higher-order and Symbolic Computation*. To appear.

[20] J. Thatcher. *Currents in the Theory of Computing*, chapter Tree automata: An informal survey. Prentice-Hall, 1973.

[21] TREX. TREX—Tree regular expressions for XML `http://www.thaiopensource.com/trex/`, 2004.

Topsl: A domain-specific language for on-line surveys

Mike MacHenry
Northeastern University
dskippy@ccs.neu.edu

Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

Abstract

There are currently few choices for social scientists who want to employ web-based surveys in their studies. They can either use a special-purpose language whose notion of flow control may be too limiting to express advanced survey designs, or use a general-purpose language that gives them the freedom to make complicated survey designs but makes them reimplement infrastructure code for saving questions to disk, generating HTML, and so on with each new survey. In this paper, we introduce Topsl, a domain-specific language embedded in PLT Scheme that takes the middle road, giving survey authors a way to reuse survey infrastructure for new surveys while also allowing them to express complicated survey designs easily.

1 Introduction

As social scientists have become more aware of the practical and theoretical benefits of gathering information online [2], the demand for web-based surveys has grown significantly in recent years. Unfortunately, technology has not improved to meet this demand. Social scientists want to design surveys that interact with participants in complicated ways that current survey languages are not capable of expressing.

Existing domain-specific languages (DSLs) for on-line surveys such as SuML [1] and QPL [10] have a limited notion of control flow. In all domain-specific survey languages the authors have found, the fundamental notion of the flow from one piece of a survey to the next is built-in and unchangeable. That means that while simple surveys are easy to implement, if a programmer wants to add a seemingly minor extension that affects the survey's flow, he or she may find the task impossible.

For example, the authors were introduced to this problem by Dr. Eli Finkel, assistant professor of psychology at Northwestern University, in the fall of 2003. Finkel found that while a plethora of contracting companies thought it would be technically feasible to loop over a user-provided input list if the user provided all items at once, none could provide the same looping facility if the user provided the items one at a time over multiple survey sessions.[1]

When studies run into these limitations, programmers resort to implementing them in a general-purpose language (GPL) such as PHP or Perl that allow them to express anything they want (as evidence that this is a popular approach, Fraley recently published a how-to guide on the subject for psychologists [4]). Unfortunately, if they make that choice, they become responsible for handling HTML generation, CGI, and data storage, all of which is unrelated to the specific survey being written. In the authors' direct experience, on-line surveys are plagued by bugs in this non-domain-specific code. For instance, in one case an on-line sociology survey implemented from scratch had a bug in its answer-saving routines that caused it to lose a significant portion of answers. When the bug was discovered, the researchers had to contact all the participants and ask them to fill the survey out again; only a fraction of the participants actually did. Such incidents, though common, are an unacceptable risk in expensive research.

It is natural that these two general strategies for solving the survey problem should emerge. Survey programs exist to collect answers to questions that will then be put into rows in a database or analyzed by a statistics program, and that might be printed out for copy-editing or for handing out to off-line survey participants. To make those operations possible, all the questions a particular survey could ask must be statically identifiable. Of course if a survey program had complete freedom at runtime to generate questions, that identification would be impossible. So, the problem must be made easier, and two simple ways to make it easier are to restrict the language in which programmers write surveys to the point where questions are statically identifiable, or restrict analysis to one particular survey and do the analysis by hand.

Both available options have serious problems, though: current DSLs afford too little flexibility in their models of flow control, and GPLs make programmers implement substantial amounts of non-domain-specific code for each survey. In this paper, we demonstrate a way to take the middle path with Topsl, a domain-specific language for writing on-line surveys embedded into the general-purpose language PLT Scheme. We arrange the embedding so that programmers can write survey code without having to worry about non-survey-specific concerns, but can use the full power of PLT Scheme when it becomes necessary.

We begin by explaining our design goals for the language, then

[1] Since survey authoring companies use similar in-house survey creation software they suffer from the same limitations.

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

present the language's syntax and semantics, and then discuss its implementation. Afterwards we discuss how the unique features of Scheme and PLT Scheme proved useful to our implementation, assess our results and point out some directions for future work.

2 Design Goals

Topsl's primary design goal is to provide a layer of abstraction that allows programmers to express surveys clearly and without having to write non-domain-specific code while still being expressive enough to write surveys with novel control-flow elements. Further, it should be easy-to-use in the sense that valid programs cannot fail to save data or otherwise malfunction and a program's final product, a series of answers, should be immediately ready for analysis in regular statistics programs.

2.1 Interoperability

One major motivation for Topsl is to eliminate the need for a programmer to write code that connects the fundamentally novel aspects of a particular survey to the technologies that manage presenting that survey to participants and saving their responses. To that end, the language must make that management invisible and unbreakable; it must not require any special action on the programmer's part for a survey to appropriately interact with participants and save results.

Topsl also needs to interface with existing technology used by social scientists in order to be useful. When social scientists conduct surveys, they store results in statistics program databases to analyze results. Importing results into a statistics program requires prior knowledge of the questions since a column will need to be created in the database for each possible question. So when social scientists design on-line surveys, they need the systems they use to create static summaries of their surveys, containing a set of all possible questions. Each question needs an identifier consistent with the dynamic implementation and question text with possible place holders where dynamic information will be filled in at runtime. Topsl must ensure that all surveys can be analyzed to produce a static summary.

2.2 Expressiveness

Handling web-programming details and interfacing with statistics programs is not enough. Social science surveys commonly use complex designs with control flow such as that present in Finkel's survey. For instance, the authors recently heard of a novel survey design in which husband and wife pairs participated cooperatively. Within each pair, one participant's answers affected the questions the other was asked. Topsl should allow programmers to express such complicated calculations using full Scheme where necessary.

In his paper, "A Universal Scripting Framework" [11], Shivers advocates the embedding of domain-specific languages within general-purpose languages to increase the expressiveness of the DSL. Topsl needs to allow programmers to "break out of the [DSL] in order to express complex computations in a [GPL]" so that complicate surveys are possible to write.

Unfortunately, allowing full Scheme in Topsl programs is at odds with our goal of automatically generating static summaries. Allowing the programmer to generate surveys using arbitrary Scheme makes statically listing all possible questions undecidable for many surveys, so some compromise is inevitable. However, since in practice even surveys with exotic designs have an obvious and predictable set of questions that will be asked, Topsl can realistically enforce a restriction that all surveys written in the language be analyzable to produce a static summary while still letting programmers write surveys with complicated control flow.

2.3 Language Growth

Languages that try to guess up-front everything their users will ever want to do tend to find that they've guessed wrong. In "Growing a Language," Steele writes that "...a main goal in designing a language should be to plan for growth" [7]. Topsl programmers should be able extend the language with new features to meet these needs. Just as in Scheme, Topsl users should be able to extend Topsl's syntax and add new functions as necessary — in addition, they should be able to add new question types and formatting options if need be. Topsl must be carefully designed to let programmers create these extensions in such a way that surveys using them remain statically analyzable.

3 Design

A Topsl survey contains a definitions context and any number of Topsl forms which make up the survey's control flow. The most basic Topsl form is the page, which is constructed from any number of page elements. The most basic page element is the question (written **?**). Questions take a question type and any number of strings or variables that make up the question text. Topsl provides basic question types like *free* (for free response questions) and *yes-no* as well as functions that produce new question types like *radio* and *multi-select* for selecting from a list of responses. A simplified Topsl syntax is as follows:

survey	::=	<definition>* <survey-element>*
definition	::=	(**define** <variable> <scheme>)
survey-element	::=	(**page** <page-element>*)
page-element	::=	(**?** <question-type> <question-text>*)
question-type	::=	<variable>
question-text	::=	<string> | <variable>

We present Topsl as a series of example surveys, augmenting the syntax with new forms as examples become more complicated. We start with a trivial example survey and show how to add abstraction and control flow to the language while still retaining the ability to generate a static summary.

3.1 A Simple Survey

We can use **page** and **?** along with a few question types to create a simple example of a complete Topsl program. In the following example, we define a new question type, *enjoy*, and then create two pages containing two questions each.

```
(define enjoy (radio "A lot" "Some" "Not at all"))
(page (? free "Where did you go to high school?")
      (? enjoy "How did you like it?"))
(page (? free "Where did you go to college?")
      (? enjoy "How did you like it?"))
```

When a participant visits the web page associated with this survey, the survey will display a page containing the appropriate questions (see figure 1). When the participant fills out the form and submits it, a second similar page will be presented, followed by a page indicating that the survey is over.

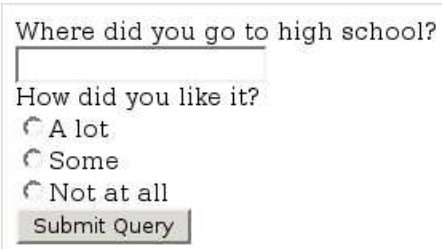


Figure 1. Example minimal HTML from a Topsl survey

Every time a participant fills out a Topsl survey, Topsl creates a response file where that participant's responses will be stored. Response files are a partial mapping from question identifiers (determined statically using the survey's static summary) to responses that represents the answers to all questions that were asked during execution. Similarly, a survey's static summary is a mapping from question identifiers to question text for all questions that could possibly be asked. The static summary of the above survey would look something like:

```
'((q1 "Where did you go to high school?")
  (q2 "How did you like it?")
  (q3 "Where did you go to college?")
  (q4 "How did you like it?"))
```

The question identifiers from the static summary can then be used to create a database table or a statistics file. Using the static summary and any number of response files, Topsl can create comma-separated value files suitable for importing into most typical spreadsheet and statistics programs used by social scientists. With the static summary and a particular response file, Topsl can correlate questions and responses in a browser for easy reading. The static summary is also useful for proofreading question text and for creating printable surveys for off-line participants.

3.2 Page Construction

Since Topsl is a domain-specific language for on-line surveys rather than a web-page construction language, survey authors should be able to express their surveys as surveys and not have to use any HTML or otherwise specify presentation. To support that, Topsl's design factors the presentation of a question from the question itself by making use of values we call page elements (**?** being one example). A page element is a value with a particular meaning in the domain of web surveys (*e.g.* a question, a grouping of questions, or a block of instruction text) that contains information about how to render itself and how to determine what answers a user submitted that correspond to it. The separation is general enough that it allows page elements to represent questions that correspond directly to individual HTML form elements, questions that correspond to multiple form elements, and even question groupings that alter presentation but that do not directly correspond to form elements at all.

In addition to providing several built-in page elements, the language must allow advanced users to create their own: we cannot predict every kind of question any researcher might ever want to ask, so we should not restrict them to only the kinds of questions we thought of when we designed the language.

3.3 Abstract Pages

Surveys frequently ask sets of questions in which each question differs from the other only in very small ways. For instance, the survey in the first example asks the participant the same set of two questions twice, once asking participants whether they enjoyed high school and the next time asking whether they enjoyed college. In the survey we built for Finkel, a block of four questions asking the participant to predict how he or she would feel about a particular topic two weeks, one month, two months, and three months in the future was repeated seven times in the course of the survey with the same phrasing each time, varying only in the topic the questions addressed. In situations like these we need to be able to make an abstraction over a page parameterized over the pieces that vary. Topsl is designed to handle such situations using a form of abstraction that looks syntactically just like a normal Scheme function definition:

```
definition  ::=  ... as before ...
             |   (define (<variable>+) <survey-element>+)
```

We can now use this new **define** syntax to abstract the original example survey.

```
(define enjoy (radio "A lot" "Some" "Not at all"))
(define (where-attended school)
  (page (? free "Where did you go to " school "?")
        (? enjoy "How did you like it?")))
(where-attended "high school")
(where-attended "college")
```

Here *where-attended* is an abstraction over a page parameterized over the school to ask the participant about — it takes a school as input and produces a page as output. The definition of *where-attended* can make use of variables in question text, and the parameterized page can be applied to arguments multiple times to yield multiple different pages. However, despite appearances, **define** in Topsl is not the same as the normal Scheme **define** and does not bind *where-attended* to a normal Scheme procedure. Instead, it defines it as a Topsl procedure that runs at compile-time rather than at run-time, and can only be used in contexts that accept Topsl.

3.4 Dynamic Surveys

In order to write surveys whose question responses affect survey control-flow we need a way to access the question responses while the survey is still running. Topsl allows this with another page element, **?/named**, that allows a programmer to name a question, and **bind**, which binds the response of a named question to an identifier which is accessible by other Topsl code. The **bind** form takes a list of identifiers to bind to named question responses and a page in which the named questions can be found.

We have now seen two ways of displaying pages in example surveys: simple page expressions constructed with the **page** form, and applications, which apply a parameterized page to arguments. To allow both mechanisms of page displaying in **bind** expressions we lift the **page** syntax into a new production, *page-expr*, and make it a new survey element:

```
survey-element  ::=  ... as before ...
                |    <page-expr>
                |    (bind (<variable>*) <page-expr>)
page-expr       ::=  (page <page-element>*)
                |    (<variable> <scheme>*)
```

The **?/named** form is identical to **?** except that it takes an additional identifier as its first argument which will be used as the question's name, significant only to **bind** and to the survey's static summary (which uses it as the question's reported name rather than automatically generating a name for it).

The **bind** form displays the page in its page expression to the participant and then binds the given names to the values the participant supplied in any subsequent expressions. The page is required to provide at least those names extracted by bind. The behavior of questions that exist on the page but which are not answered by the participant is specified by the question type. For this paper, we use only mandatory question types, which Topsl requires the participant to answer or it will redisplay the page. We can use the **?/named** and **bind** forms to construct the following survey:

```
(bind (fav-num)
  (page (?/named fav-num free "What's your favorite number?")))
(page (? free "Why do you like " fav-num "?"))
```

This survey asks a participant two questions, the second of which has text that cannot be determined until the first one is answered. When this survey is visited on the web it will ask the participant for his or her favorite number. When the page is submitted, the next page will have one question which will ask the participant why they like the number he or she submitted, which will be contained in the question text.

Clearly Topsl cannot know the complete text of dynamically-determined questions when generating a static summary. In such cases, the static summary uses the identifier in the question text as a placeholder for the dynamic value. For instance, the static summary of the above survey looks like this:

```
'((fav-num "What is your favorite number?")
  (q2 "Why do you like " fav-num "?"))
```

3.5 Adding Control Flow

If we did not need to support static summaries, then making the **?**, **page** and related Topsl forms behave exactly as normal functions would be ideal. However, were we to make that design decision, static summaries would be impossible to build. For instance, the following program would be legal even though the number of questions on any given page cannot be determined:

```
(define (problem n)
  (if (zero? (random 2))
      '()
      (cons (? yes-no "Is " n " prime?")
            (problem (add1 n)))))
(apply page (problem 0))
```

We could use a static-flow analysis algorithm such as the set-based analysis (SBA) system developed by Meunier *et al* to indicate what values could possibly flow into questions and pages [9], but even the best of those techniques are too conservative for our needs. Using the values obtained from SBA to construct a static summary would generate possibly infinite number of questions that would never actually appear in the survey, making it impossible to construct the static summary in some cases.

To be able to generate static summaries reliably, we restrict the syntax of Topsl so that the contents of every page and every question a Topsl program will display is syntactically apparent (perhaps with information that does not affect the number of questions on a page), and let the static summary include all questions that appear syntactically in the program. This restriction does not eliminate all analysis errors since questions that a Topsl program can never reach will be included in its static summary, we believe errors of that nature will not be important in practice.

The biggest impact of that restriction is that Topsl code cannot intermingle arbitrarily with Scheme code. To ameliorate that situation, Topsl includes its own control-flow forms that enforce syntactic restrictions to allow for analysis but give programmers significant power over the flow of their surveys. We will show two examples of control-flow forms, **when** and **for-each**:

```
survey-element  ::=  ... as before ...
                |    (when <scheme> <survey-element>+)
                |    (for-each <variable> <scheme>+)
```

Both forms behave similarly to their Scheme namesakes. If the test position of the **when** form is a true Scheme value then the consequent Topsl expressions are executed. The **for-each** form takes a variable which must be defined as a parameterized page. All subsequent Scheme expressions must evaluate to list values whose elements will be used as arguments to the parameterized page. For instance, arbitrary Scheme is allowed in the test position of Topsl's **when** form, but the consequents must be Topsl forms. This restriction allows us to easily extract the static summary by listing the questions found in all control paths without having to perform static evaluation of full Scheme to determine what those paths could be.

We can now construct a more interesting survey where the responses affect control flow. The survey in figure 2 uses the multi-select function to create a new question type *countries* which will allow the participant to select any number of countries from an HTML selection box. We then define a page, *about*, which takes a string, *country*, and asks a participant the questions we are interested in. The survey then uses the **bind** form to bind two variables, *england?*, a boolean value from the yes-no question, and *been-to*, which is a list of strings selected from the multi-select box. The survey uses **when** to ask the participant about England if *england?* is true and then loops over the other countries the participant has been to, asking about those.

3.6 Finkel's Loop and the Typical Expression

In the previous example we loop over the return value of a *multi-select* box, but Topsl also needs to be able to loop over values generated from arbitrary computations such as lookup in a database.[2] We support this feature by allowing programmers to define arbitrary Scheme functions and apply them in control-flow forms where

[2]It was this specific feature that professional survey authoring companies could not provide Finkel.

```
(define countries (multi-select "Germany" "France" "Spain"))
(define (about country)
  (page (? yes-no "Did you like " country "?")
        (? yes-no "Is it cold in " country "?")))
(bind (england? been-to)
  (page (?/named england? yes-no "Have you ever been to England?")
        (?/named been-to countries "Where else have you been to?")))
(when england? (about "England"))
(for-each about been-to)
```

```
'((q1 "Have you ever been to England?")
  (q2 "Which other countries have you been to?")
  (q3 "Is it cold in England?")
  (q4 "Did you like England?")
  (q5 "Is it cold in " country "?")
  (q6 "Did you like " country "?"))
```

Figure 2. A complete Topsl survey with static summary

Scheme is allowed. In the following example we use a Scheme function, *get-all-other-countries*, which takes the list of countries the participant was asked about above and returns all the other countries found in a database.

```
;; definition of get-all-other-countries elided
(for-each about (get-all-other-countries been-to))
```

We have now shown both extremes of how a Topsl programmer can use Topsl forms to control flow. In the simple example the programmer branched and looped over the responses of questions. In the most recent example the programmer wrote a complicated Scheme program to loop over. In the authors' experience, the complexity of most conditionals fell somewhere in the middle. It was particularly common to have a nested boolean expression with a few common Scheme predicates on the question results. For example, the code snippet below can be used to decide whether the subject has a passport and then ask pertinent questions. For the sake of example, the authors assume that if a participant has been to more than one country then he or she has a passport.

```
(when (or (> (length been-to) 1)
          (and england? (not (empty? been-to))))
  (page (? free "When did you get your passport?")))
```

Allowing simple expressions like the one in the predicate position above gives Topsl a simple learning curve. It is not necessary for Topsl programmers to understand full Scheme, just the functions they want to use. As a result a Topsl programmer's knowledge can scale with his or her increasingly complex surveys.

3.7 Growing Topsl

Topsl provides the **require** from, taken from mzscheme, which allows Topsl programs to import other modules containing new Topsl forms. The modules the surveys require can be written in Topsl but it is not required. As such programmers can extend the Topsl language in languages other than Topsl itself, including full Scheme. In doing so, the author of that module makes a trade off. He or she gives up the safety ensured by Topsl forms and must take on the responsibility of ensuring that static analysis is maintained and that the new core forms behave as expected. However, he or she is no longer restricted to what Topsl is able to express to create new forms. This mechanism for extending a language using another language, pointed out by Krishnamurthi [8], provides Topsl with unbounded expressibility even the in presence of the static analysis restriction, allowing the language to grow in ways not anticipated by its authors.

4 Implementation

Implementation of the "dynamic" portion of Topsl — that is, the portion that presents web pages to a user and stores that user's responses — is reasonably straightforward. The user's program becomes a servlet for the PLT web server [6]. The server handles session management and other HTTP-related issues and allowed us to think of the web as a normal input/output device, greatly simplifying our development effort. Topsl programs are the composition of Scheme macros and functions that generate XHTML pages encoded as S-expressions and hand them off to the PLT web server for shipping to the users. We prevent programmers from using other Scheme code in arbitrary positions with the PLT module system, which allows us to provide an alternate language for programs.

While this approach works very well, it does have one important problem. The semantics of the PLT web server do not exactly match those we need for Topsl programs: we found that a user answering a question was most naturally modeled with a destructive update to a global record, with the caveat that if a participant hits the back button the answer is erased. Unfortunately, the PLT server does not undo destructive updates to variables when a user hits the back button. However, it does restore the values of lexically-bound variables, so we solved the problem by principled use of state-passing style in our implementation: a record representing the current answers to all questions is passed in to every function that needs to read or alter them. After the survey is completed, the Topsl framework passes this "result monad" to the data-storage module, which writes it out. We were initially worried that this strategy might be too memory-intensive on the server and that an approach in which the framework immediately stored all answers in a database would be necessary, but in practice even our relatively modest server (a Pentium III-800 MHz with 128MB RAM) handled the load with no problems.

Implementation of the static summary feature turns out to be mostly trivial as a result of restricting Topsl's syntax to only Topsl forms. In a survey without abstraction, static analysis is trivial, the summary is essentially just the syntax of that survey minus any Scheme expressions. Allowing parameterized pages means we need to statically expand any parameterized pages where they are applied. To enable this, Topsl provides a special **define** form which behaves differently from the normal Scheme behavior when defining Topsl values. The **define** form expands its body and checks to see if that body expands to a Topsl core form. If it does, the **define** expands to a **define-syntax**. If not, it expands to a regular Scheme **define**. Macros cannot be higher-order; however, parameterized pages are only available from within Topsl, and the Topsl forms such as **for-each** that treat pages as higher-order functions can be written to cope with the altered interface.

5 What Does Scheme Give Us?

Scheme has a powerful macro system that allows us to write Topsl forms in terms of Scheme in a very clear and easy-to-maintain manner while avoiding the need to write a parser and compiler from scratch. Scheme's macro system also provides us two very important additional advantages. First, since Topsl is defined as a collection of macros over Scheme, we get seamless escapes into Scheme without any complications. That is, we do not have to marshal data or define a communication "wrapper" layer for communicating between Topsl and Scheme: under the hood, it's all just Scheme. Second, extending the language with new syntactic forms is a simple process of defining a macro over existing Topsl forms. That allows us to grow our language to meet the unforeseen requirements of future surveys without having to edit the Topsl compiler.

6 What Does PLT Scheme Give Us?

The PLT Scheme suite provides two tools that make our work easier: the PLT module system and the PLT web server.

The PLT module system gives us a flexible way to build languages from other languages [3]. Writing Topsl as a module language gives us the ability to compile Topsl code in any way we choose, taking an entire program at once rather than dealing with one subexpression at a time as we would have to with normal macros. It also allowed us to reuse existing Scheme code in our implementation without having to handle name space collisions.

The continuation-based PLT web server [6] made it much easier for us to make language constructs that query a web user for input. Topsl is an imperative language where presenting a page to a participant is implemented as a call to a function that returns the participant's answers and presents the page to that participant as a side effect. The PLT web server allows us to ignore the complication of web-based communication and implement that feature in a natural way, without worrying about implementing the complicated transformations that would otherwise be necessary to make it work properly [5].

7 Experience

Topsl's first application, and our motivation for developing it, was an on-line survey used in a longitudinal study of dating relationships at Northwestern University. The survey had 70 participants each of whom was asked to visit the survey site once every two weeks and answer some subset of the survey's 248 unique questions that depended on his or her answers from all previous sessions and from the current one: for instance, if the participant had reported that they started dating someone in one session and said they were single in the next, the survey would proceed to a page of questions about the breakup. Also, every time a participant broke up with someone, that person's initials were added to a list; on every subsequent session the survey would present a few questions about each person on that list.

We found Topsl to be an invaluable asset in developing the survey. It let us focus on the survey's particular unique features without needing to worry about our changes introducing bugs in the underlying mechanisms that handled sessions and data storage. For that reason, we were able to develop the survey extremely rapidly given its complexity: we developed prototypes of both the language and the survey in two days, and afterward we were able to modify the survey easily to suit the various revisions its designer requested. For instance, one early revision requested was that we randomize the order in which questions in certain groups were presented to participants. We accommodated that request by writing a Topsl extension that introduced a new page element that randomly shuffled its sub-elements when it presented them on-screen. Data storage and other aspects of presentation were unaffected, so we were able to make the change and be confident of its functionality in only a few hours.

The static summary technique discussed in this paper was developed as a result of failings in that prototype. Our original design required giving every question a unique name and further required a redundant listing of that name in some situations. We found the burden of naming each question quickly became a maintenance nightmare: a request to insert or remove a question would ruin our naming strategy, and changing a name in one place but not another would cause apparent data loss. The survey summary technique avoids this problem while still giving us more than enough flexibility to implement our original survey and every other survey we have seen since.

8 Related Work

There are a considerable number of mechanisms for creating online surveys apart from implementing them in a general-purpose language. Two domain-specific languages, SuML and QPL, stand out as being the closest to the goals the authors set for Topsl.

SuML is an XML/Perl-based survey language in which the programmer describes a survey in an XML document which follows the SuML Schema. The SuML Schema has a **question** element which contains question text and a sequence of allowable responses, much like Topsl. The root **survey** element contains any number of questions and a **routing** element that describes control flow. The **routing** element contains any number of **if** and **ask** elements which are composed to ask questions in the survey and branch on their responses.

The programmer creates two files in addition to the content of the survey: an XSLT stylesheet and a front-end Perl CGI program. The style sheet is responsible for describing what a survey will look like when presented on the web to a participant, and multiple stylesheets can be written for different mediums. The front-end is a Perl CGI program that acts as the entry point to the survey.

SuML's most significant problem for our purposes is its notion of control-flow is very limited, providing its users with only an **if** statement with which to branch to different parts of the survey. Furthermore, the test position of the **if** is written in language for accessing various fields of the XML allowing the programmer to reference question responses. This approach limits control flow to being affected by only responses given in the current survey execution.

In addition, SuML is somewhat too generic for our purposes. The user-written Perl CGI is in charge of driving the survey by using SuML's Perl API to get the next questions to be asked and then present them as well as storing the results of the questions asked. Putting the burden on the programmer makes survey development more difficult, time-consuming, and error-prone.

QPL is another domain-specific language for creating surveys that suffers from very similar problems to SuML's. QPL's semantics are reminiscent of BASIC: it is an imperative language using **if** and **goto** for control flow along with a large set of built-in predicates

used for conditional testing. Current distributions provides users with a large set of comparison functions for use with **if**; however, it lacks an means of growing to meet programmers' changing needs.

9 Further Work

One major avenue of future work we plan on pursuing is making it easier for non-programmers to develop simple surveys in Topsl. While programmers who want a rapid way to develop surveys and are experienced with Scheme should find Topsl intuitive, social scientists who have no programming experience may have difficulty with it. To that end, we suspect that providing a graphical syntax with a WYSIWYG page construction to make the syntax more like word processing would make Topsl more natural for social scientists. Syntax for forms like **when** and **for-each** have been taken from Scheme to meet our programmer audience's expectations of how they should be used, but a graphical syntax that relates pages with flow-control arrows would be more natural for non-programmers. We expect to be able to implement this syntax with the help of PLT Scheme's MrEd toolkit and the substantial graphical editing features of DrScheme.

We would also like to investigate the possibility of adding shared question and page libraries to Topsl. Since social scientists often include the same questions verbatim on multiple surveys to make the surveys more easily comparable, shared libraries are a natural fit. However, they pose some interesting problems: with our current design, for instance, every question whose answer is important to a survey's flow control must be named explicitly in its declaration. In a library, this would not work out well since library authors would have to give every question a name (which is impractical in our experience) or guess which questions will be important to future surveys (which would force users to copy the library and make source code modifications if the library author guesses wrong). A solution to this problem would be quite useful, so we consider it an important topic to investigate.

10 Contributions

We have designed and implemented a survey language system that uses Scheme's capacity to build new languages to solve a pressing problem in many of the social sciences. In the process, we have illustrated the power of building new languages to simultaneously make programs easier to write and less error-prone.

11 Acknowledgments

The authors would like to thank Matthias Felleisen for invaluable guidance throughout the development of this project.

12 References

[1] Barclay, Lober, Huq, Dockery, and Karras. SuML: A survey markup language for generalized survey encoding. In *AMIA Annual Symposium*, 2002.

[2] Michael Birnbaum, editor. *Psychological Experiments on the Internet*. Academic Press, 2000.

[3] Matthew Flatt. Composable and compilable macros. In *ICFP*, October 2002.

[4] R. C. Fraley. *How to conduct behavioral research over the Internet: A beginner's guide to HTML and CGI/Perl*. Guilford, 2004.

[5] Graunke, Findler, Krishnamurthi, and Felleisen. Automatically restructuring programs for the web. In *Automated Software Engineering*, 2001.

[6] Graunke, Krishnamurthi, Van der Hoeven, and Felleisen. Programming the web with high-level programming languages. In *ESOP*, 2001.

[7] Guy L. Steele Jr. Growing a language. *Journal of Higher-Order and Symbolic Computation*, 12:221 – 236, October 1999.

[8] Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, May 2001.

[9] Meunier, Findler, Steckler, and Wand. Selectors make analysis of `case-lambda` too hard. In *Scheme and Functional Programming*, 2001.

[10] U.S. General Accounting Office. QPL. Software: http://www.gao.gov/qpl/.

[11] Olin Shivers. A universal scripting framework or lambda: the ultimate 'little language'. *Concurrency and Parallelism, Programming, Networking, and Security, Lecture Notes in Computer Science*, 1179:254–265, 1996.

Lexer and Parser Generators in Scheme

Scott Owens Matthew Flatt
University of Utah

Olin Shivers Benjamin McMullan
Georgia Institute of Technology

Abstract

The implementation of a basic LEX-style lexer generator or YACC-style parser generator requires only textbook knowledge. The implementation of practical and useful generators that cooperate well with a specific language, however, requires more comprehensive design effort. We discuss the design of lexer and parser generators for Scheme, based on our experience building two systems. Our discussion mostly centers on the effect of Scheme syntax and macros on the designs, but we also cover various side topics, such as an often-overlooked DFA compilation algorithm.

1 Introduction

Most general-purpose programming systems include a lexer and parser generator modeled after the design of the LEX and YACC tools from UNIX. Scheme is no exception; several LEX- and YACC-style packages have been written for it. LEX and YACC are popular because they support declarative specification (with a domain-specific language), and they generate efficient lexers and parsers. Although other parsing techniques offer certain advantages, LEX- and YACC-style parsing remains popular in many settings. In this paper, we report on the design and implementation of LEX- and YACC-style parsers in Scheme. Scheme's support for extensible syntax makes LEX- and YACC-style tools particularly interesting.

- Syntax allows the DSL specifications to reside within the Scheme program and to cooperate with the programming environment. We can also lift Scheme's syntactic extensibility into the DSL, making it extensible too.
- Syntax supports code generation through a tower of languages. Breaking the translation from grammar specification to Scheme code into smaller steps yields a flexible and clean separation of concerns between the levels.

Additionally, lexer and parsers are examples of language embedding in general, so this paper also serves as an elaboration of the "little languages" idea [13].

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

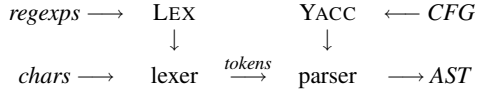
We base our discussion on two parsing tools, PLT and GT, and present specific design notes on both systems along with discussion on why certain ideas work well, and how these systems differ from others. The PLT system most clearly demonstrates the first point above. PLT's novel features include an extensible regular expression language for lexing and a close interaction with the DrScheme programming environment. The GT system most clearly illustrates the second point. In GT, a parser's context-free grammar is transformed through a target-language independent language and a push-down automaton language before reaching Scheme, with potential for optimization and debugging support along the way.

2 Background

We briefly describe the essential design of LEX and YACC and how it can fit into Scheme. We also discuss how the existing LEX- and YACC-like systems for Scheme fit into the language.

2.1 LEX and YACC

A text processor is constructed with LEX and YACC by specifying a lexer that converts a character stream into a stream of tokens with attached values, and by specifying a parser that converts the token/value stream into a parse tree according to a context-free grammar (CFG). Instead of returning the parse tree, the parser performs a single bottom-up computation pass over the parse tree (synthesized attribute evaluation) and returns the resulting value, often an abstract-syntax tree (AST). LEX generates a lexer from a regexp DSL, and YACC generates a parser from a CFG DSL.



From a programmer's perspective, building a text processor takes three steps:

- Creation of regular expressions (regexps) that describe the token structure. These are typically defined outside the lexer with a regexp abbreviation facility.
- Creation of the lexer by pairing the regexps with code to generate tokens with values.
- Creation of the parser with a CFG that describes the syntax, using token names as non-terminals. Attribute evaluation code is directly attached to the CFG.

A lexer is occasionally useful apart from a parser and can choose to produce values other than special token structures. Similarly, a

Figure 1 Lexical scope in a lexer

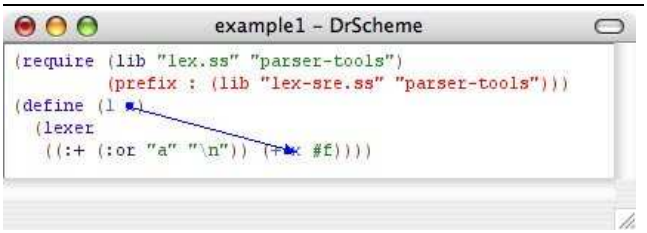
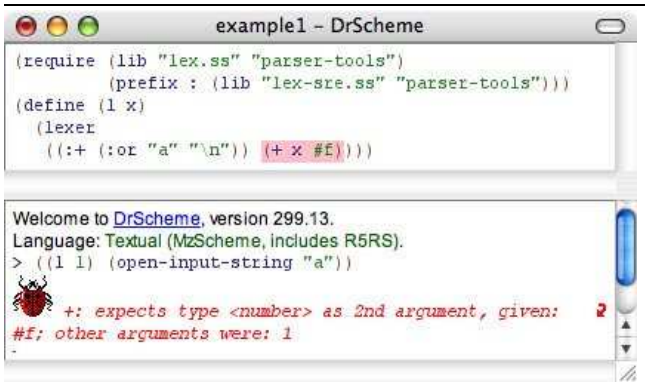


Figure 2 Source highlighting of a runtime error



parser can operate without a lexer or with a hand-written lexer that returns appropriate token structures. Nevertheless, LEX and YACC (and the lexers and parsers they generate) are used together in most cases.

Operationally, LEX converts the regexps into a deterministic finite automaton (DFA) and YACC converts the CFG into a push-down automaton (PDA). These conversions occur at lexer and parser generation time. The DFA can find a token in linear time in the length of the input stream, and the PDA can find the parse tree in linear time in the number of tokens. The transformation from regexps and CFG can be slow (exponential in the worst case), but performing these computations once, at compile time, supports deployment of fast text-processing applications.

2.2 The Scheme Way

UNIX LEX and YACC operate in a file-processing batch mode. They read a lexer or parser specification from an input file and write a program to an output file. With batch compiled programming languages, *e.g.*, C or Java, this is the best that can be done. The build system (such as a makefile) is told how to convert the specification file into a code file, and it does so as needed during batch compilation.

The file-processing approach does not fit naturally into Scheme's compilation model. Instead of using an external batch compilation manager, most Scheme programs rely on a compilation strategy provided by the language implementation itself. The simplest way to cause these compilation managers to execute a Scheme program is to package it in a macro. The compilation manager then runs the program while it macro-expands the source. Specifically, when a lexer or parser generator is tied into the Scheme system via a macro, the macro expander invokes the regexp or grammar compiler when the internal compilation system decides it needs to. Each of the PLT and GT parser tools syntactically embeds the lexer or parser specification inside the Scheme program using lexer and parser macros. This solution easily supports LEX- and YACC- style pre-computation without breaking Scheme's compilation model.

With a macro-based approach, a lexer or parser specification can appear in any expression position. Hygiene then ensures that variables in embedded Scheme expressions refer to the lexically enclosing binding (see Figure 1). Furthermore, the embedded code automatically keeps source location information for error reporting, if the underlying macro expander tracks source locations as does PLT Scheme's (see Figure 2). A stand-alone specification achieves neither of these goals easily.

Source-location tracking and lexical scoping lets refactoring tools, such as DrScheme's Check Syntax, assist programmers with their parser and lexer actions. Check Syntax draws arrows between statically apparent binding and bound variable occurrences and can α-rename these variables. It also provides assistance with the declarative part of a parser specification, correlating non-terminals on the right and left of a production and correlating uses of `$n` attribute references with the referent terminal or non-terminal.

2.3 Previous Work

We categorize the many lexer and parser generators written in Scheme as follows: those that do not use s-expression syntax, those that use s-expressions but do not provide a syntactic form for integrating the specifications with Scheme programs, and those that do both.

Danny Dubé's SILex lexer generator [6] fits in the first category, closely duplicating the user interface of LEX. Mark Johnson's LALR parser generator [10] and the SLLGEN system [9] fall into the second category. Both provide a function that consumes a list structure representing a grammar and returns a parser. With Scheme's quasiquote mechanism, programmers can write the grammar in s-expression format and generate grammars at run time. This approach sacrifices the efficiency of computing the parser from the grammar at compile time and also hampers compile-time analysis of the CFG and attached code.

Dominique Boucher's original lalr-scm system [3] falls into the second category, encoding the CFG in an s-expression. It uses DeRemer and Pennello's LALR algorithm [5] to process the grammar and, unlike previous tools, supports ahead-of-time compilation of the CFG. However, it does not integrate specifications into Scheme via macros, instead it provides a compiler that maps CFG s-expressions to a parse table. The table is printed out and incorporated into source code along with an interpreter to drive the parser. In this sense, the parser development cycle resembles YACC's. Boucher's original implementation is also missing some important functional elements, such as associativity declarations to resolve shift/reduce ambiguities.

Design particulars aside, Boucher's source code was influential, and his complete, debugged, and portable LALR implementation provided a foundation for several later efforts in our third category. Serrano's Bigloo Scheme system [11] incorporated Boucher's implementation, with extensions. Critically, Bigloo uses macros to embed the parser language directly into Scheme. (Bigloo also supports embedded lexers.) Shivers and his students subsequently

adopted Serrano's code at Georgia Tech for compiler work, and then massively rewrote the code (for example, removing global-variable dependencies and introducing record-based data structures for the ASTs) while implementing the GT parser design described in this paper. Boucher has addressed the above concerns, and the current lalr-scm system supplies a form for incorporating a CFG in a program.

Sperber and Thiemann's Essence parser generator [16] also falls into the third category, using an embedded s-expression based CFG definition form. Instead of a YACC-style CFG to PDA compilation step, Essence uses partial evaluation to specialize a generic LR parser to the given CFG. The partial evaluation technology removes the need for a separate compilation step, while ensuring performance comparable to the YACC methodology. Our use of macros lets us take a compilation-based approach to implementation—a simpler and less exotic technology for achieving performance.

3 Regular Expressions

Defining regular expressions that match the desired tokens is the first step in creating a lexer, but regexps are also used as a pattern language in many other text processing tools. For example, programming languages often support regular expression matching as a computational device over strings. Hence we first consider regexps by themselves.

3.1 Notational Convenience

Many regexp matching libraries use the POSIX regexp syntax embedded in a string. This approach requires the insertion of escape characters into the POSIX syntax (a veritable explosion of \ characters, since \ is used as the escape character in both POSIX and Scheme). An s-expression based regexp language fits more naturally into Scheme and can still include a form for string-embedded POSIX syntax, if desired.

SREs [14], Bigloo Regular Grammars [11, section 9], and PLT lexer regexps all use s-expression syntax. The SRE notation is oriented toward on-the-fly regexp matching functions and was developed for the scsh systems programming environment [12, 15]. Bigloo Regular Grammars are designed for lexer specification, as are the PLT lexer regexps. The PLT Scheme lexer generator uses the syntax of Figure 3,[1] and SREs use the syntax of Figure 4. Bigloo uses notation similar to SREs without the dynamic unquote operation.

3.2 Abstraction

To avoid unnecessary duplication of regular expressions, a regexp language should support abstraction over regular expressions. Consider the R^{5}RS specification of numbers:

$$\langle uintegerR \rangle \rightarrow \langle digitR \rangle^{+}\#^{*}$$

This example suggests naming a regexp, such as *digit8* for the digits 0 to 7, and building a regexp producing function *uinteger* that takes in, for example, *digit8* and produces the regexp *uinteger8*.

[1]The regexp language described in Figure 3 is new to version 299.13 of PLT Scheme. The syntax of versions 20x does not support the definition of new forms and is incompatible with other common s-expression notations for regexps.

In systems that support runtime regexp generation, the abstraction power of Scheme can be applied to regexps. String-based regexps support run-time construction through direct string manipulation (*e.g.*, `string-append`). The SRE system provides constructors for SRE abstract syntax, allowing a Scheme expression to directly construct an arbitrary SRE. It also provides the `(rx` *sre* `...)` form which contains s-expressions compiled according to the SRE syntax. Think of `rx` in analogy with `quasiquote`, but instead of building lists from s-expressions, it builds regexps from them. The `unquote` form in the SRE language returns to Scheme from SRE syntax. The Scheme expression's result must be a regexp value (produced either from the AST constructors, or another `rx` form). The R^{5}RS example in SRE notation follows.

```
(define (uinteger digit)
  (rx (: (+ ,digit) (* "#"))))
(define (number digit)
  (rx (: (? "-") ,(uinteger digit)
         (? ".") (? ,(uinteger digit)))))
(define digit2 (rx (| "0" "1")))
(define digit8 (rx (| "0" ... "7")))
(define number2 (number digit2))
(define number8 (number digit8))
```

A lexer cannot use the `unquote` approach, because a lexer must resolve its regexps at compile time while building the DFA. Thus, PLT and Bigloo support static regexp abstractions. In both systems, the regexp language supports static reference to a named regexp, but the association of a regexp with a name is handled outside of the regexp language. In Bigloo, named regexp appear in a special section of a lexer definition, as in LEX. This prevents sharing of regexps between lexers. In PLT, a Scheme form `define-lex-abbrevs` associates regexps with names. For example, consider the `define-lex-abbrevs` for a simplified *uinteger2*:

```
(define-lex-abbrevs
  (digit2 (union "0" "1"))
  (uinteger2 (repetition 1 +inf.0 digit2))))
```

Each name defined by `define-lex-abbrevs` obeys Scheme's lexical scoping and can be fully resolved at compile time. Thus, multiple PLT lexers can share a regexp.

To support the entire R^{5}RS example, the PLT system uses macros in the regexp language. A form, `define-lex-trans`, binds a transformer to a name that can appear in the operator position of a regexp. The regexp macro must return a regexp, as a Scheme macro must return a Scheme program. The system provides libraries of convenient regexp forms as syntactic sugar over the parsimonious built-in regexp syntax. Of course, programmers can define their own syntax if they prefer, creating regexp macros from any function that consumes and produces syntax objects [7][8, section 12.2] that represent regexps.

Using the SRE operator names, the R^{5}RS example becomes:

```
(define-lex-trans uinteger
  (syntax-rules ()
    ((_ digit) (: (+ digit) (* "#")))))
(define-lex-trans number
  (syntax-rules ()
    ((_ digit)
     (: (? "-") (uinteger digit)
        (? ".") (? (uinteger digit))))))
```

Figure 3 The PLT lexer regular-expression language

```
re ::= ident                          ; Bound regexp reference
     | string                         ; Constant
     | char                           ; Constant
     | (repetition lo hi re)          ; Repetition. hi=+inf.0 for infinity.
     | (union re ...)                 ; General set
     | (intersection re ...)          ;    algebra on
     | (complement re)                ;    regexps
     | (concatenation re ...)         ; Sequencing
     | (char-range char char)          ; Character range
     | (char-complement re ...)       ; Character set complement, statically restricted to 1-char matches
     | (op form ...)                  ; Regexp macro
lo ::= natural number
hi ::= natural number or +inf.0
op ::= identifier
```

Figure 4 The SRE regular-expression language (some minor features elided)

```
re ::= string                 ; Constant
     | char                   ; Constant
     | (** lo hi re ...)      ; Repetition. hi=#f for infinity.
     | (* re ...)             ; (** 0 #f re ...)
     | (+ re ...)             ; (** 1 #f re ...)
     | (? re ...)             ; (** 0 1 re ...)
     | (string)               ; Elements of string as char set
     | (: re ...)             ; Sequencing
     | (| re ...)             ; Union
     | (& re ...)             ; Intersection, complement, and difference
     | (~ re)                 ;    statically restricted
     | (- re re ...)          ;    to 1-char matches
     | (/ char ...)           ; Pairs of chars form ranges
     | (submatch re ...)      ; Body is submatch
     | ,exp                   ; Scheme exp producing dynamic regexp
     | char-class             ; Fixed, predefined char set
lo ::= natural number
hi ::= natural number or #f
char-class ::= any | none | alphabetic | numeric | ...
```

```
(define-lex-abbrevs
  (digit2 (| "0" "1"))
  (digit8 (| "0" ... "7"))
  (number2 (number digit2))
  (number8 (number digit8)))
```

The `define-lex-trans` and `define-lex-abbrevs` forms are macro-generating macros that define each given name with a `define-syntax` form. The regexp parser uses `syntax-local-value` [8, section 12.6] to locates values for these names in the expansion environment. Unfortunately, many common regexp operator names, such as + and ∗, conflict with built-in Scheme functions. Since Scheme has only one namespace, some care must be taken to avoid conflicts when importing a library of regexp operators, *e.g.*, by prefixing the imported operators with `:` using the prefix form of `require` [8, section 5.2].

3.3 Static Checking

Both the SRE system and the PLT lexer generator statically check regexps. The SRE language supports a simple type inference mechanism that prevents character set operations, such as intersection (`&`), from being misapplied to regexps that might accept more or less than a single character. This system has two types: `1` and `n`.

Figure 5 Illustrative fragment of SRE type system

`T ::= 1 | n`

$$\frac{}{\vdash \mathit{char} : \mathtt{1}} \qquad \frac{\vdash re_1 : t_1 \cdots \vdash re_m : t_m}{\vdash (* \; re_1 \ldots re_m) : \mathtt{n}}$$

$$\frac{\vdash re_1 : \mathtt{1} \cdots \vdash re_m : \mathtt{1}}{\vdash (\mid re_1 \ldots re_m) : \mathtt{1}} \qquad \frac{\vdash re_1 : \mathtt{1} \cdots \vdash re_m : \mathtt{1}}{\vdash (\& \; re_1 \ldots re_m) : \mathtt{1}}$$

$$\frac{\vdash re_1 : t_1 \cdots \vdash re_m : t_m}{\vdash (\mid re_1 \ldots re_m) : \mathtt{n}}$$

Intuitively, a regexp has type `1` iff it must match exactly one character and type `n` if it can match some other number of characters. Regexps with misapplied character set operations have no type.

Figure 5 presents the type system for ∗, `&`, `|`, and *char* SREs—the rules for the polymorphic `|` are most interesting. The macro that processes SREs, `rx`, typechecks regexps that contain no `,`*exp* forms. For dynamic regexps, it inserts a check that executes when the regexp is assembled at run time.

The PLT lexer regexp language also check character set operations. Instead of using a separate typechecking pass, it integrates the computation with the regexp parsing code. Not only must the lexer generator reject mis-applications of character set primitives, but it must internally group character set regexps into a specialized character set representation. In other words, `(| "a" (| "b" "c"))` is internally represented as `(make-char-set '("a" "b" "c"))`.[2] The DFA-construction algorithm usually operates on only a few characters in each set, whereas it considers each character in a union regexp individually. Thus the character set grouping yields a requisite performance enhancement.

3.4 Summary

Even though a lexer generator must resolve regular expressions statically, its regexp language can still support significant abstractions. Syntactic embedding of the regexp language, via macros, is the key technique for supporting programmer-defined regexp operators. The embedded language can then have static semantics significantly different from Scheme's, as illustrated by the regexp type system.

4 Lexer Generator

After defining the needed regexps, a programmer couples them with the desired actions and gives them to the lexer generator. The resulting lexer matches an input stream against the supplied regexps. It selects the longest match starting from the head of the stream and returns the value of the corresponding action. If two of the regexps match the same text, the topmost action is used.

A lexer is expressed in PLT Scheme with the following form:

```
(lexer (re action) ...)
```

The `lexer` form expands to a procedure that takes an input-port and returns the value of the selected action expression.

The PLT lexer generator lacks the left and right context sensitivity constructs of LEX. Neither feature poses a fundamental difficulty, but since neither omission has been a problem in practice, we have not invested the effort to support them. Cooperating lexers usually provide an adequate solution in situations where left context sensitivity would be used (encoding the context in the program counter), and right context sensitivity is rarely used (lexing Fortran is the prototypical use). The Bigloo lexer supports left context sensitivity, but not right.

4.1 Complement and Intersection

In Section 3.2, we noted that a regexp language for lexers has different characteristics than regexp languages for other applications in that it must be static. In a similar vein, lexer specification also benefits from complement and intersection operators that work on all regexps, not just sets of characters. The PLT lexer generator supports these, as does the lexer generator for the DMS system [2].

Intersection on character sets specializes intersection on general regexps, but complement on character sets is different from complement on general regexps, even when considering single character regexps. For example, the regexp `(char-complement "x")` matches any single character except for `#\x`. The regexp `(complement "x")` matches any string except for the single character string `"x"`, including multiple character strings like `"xx"`.

The following regexp matches any sequence of letters, except for those that start with the letters b-a-d (using a SRE-like sugaring of the PLT regexp syntax with `&` generalized to arbitrary regexps).

```
(& (+ alphabetic)
   (complement (: "bad" any-string)))
```

The equivalent regexp using only the usual operators (including intersections on character sets) is less succinct.

```
(| (: (& alphabetic (~ "b"))
      (* alphabetic))
   (: "b" (& alphabetic (~ "a"))
          (* alphabetic))
   (: "ba" (& alphabetic (~ "d"))
           (* alphabetic)))
```

The formal specification more closely and compactly mirrors the English specification when using complementation and intersection. We have used this idiom to exclude certain categories of strings from the longest-match behavior of the lexer in specific cases.

As another example, a C/Java comment has the following structure: /* followed by a sequence of characters not containing */ followed by */. Complementation allows a regexp that directly mirrors the specification.

```
(: "/*"
   (complement (: any-string "*/" any-string))
   "*/")
```

The regexp `(: any-string "*/" any-string)` denotes all strings that contain */, so `(complement (: any-string "*/" any-string))` denotes all strings that do not contain */. Notice that `(complement "*/")` denotes all strings except the string `"*/"` (including strings like `"a*/"`), so it is not the correct expression to use in the comment definition. For a similar exercise, consider writing the following regexp without complement or intersection.

```
(& (: (* "x") (* "y"))
   (: any-char any-char any-char any-char))
```

4.2 Lexer Actions

A lexer action is triggered when its corresponding regexp is the longest match in the input. An action is an arbitrary expression whose free variables are bound in the context in which the lexer appears. The PLT lexer library provides several variables that let the action consult the runtime status of the lexer.

One such variable, `input-port`, refers to the input-port argument given to the lexer when it was called. This variable lets the lexer call another function (including another lexer) to process some of the input. For example,

```
(define l
  (lexer
    ((+ (or comment whitespace))
     (l input-port))
    ...))
```

instructs the lexer to call `l`, the lexer itself, when it matches whitespace or comments. This common idiom causes the lexer to ignore

[2] To handle the large character sets that can arise with Unicode codepoints as characters, the character set representation is actually a list of character intervals.

whitespace and comments. A similar rule is often used to match string constants, as in

```
(#\" (string-lexer input-port))
```

where `string-lexer` recognizes the lexical structure of string constants.

The `lexeme` variable refers to the matched portion of the input. For example,

```
(number2 (token-NUM (string->number lexeme 2)))
```

converts the matched number from a Scheme string into a Scheme number and places it inside of a token.

A lexer often needs to track the location in the input stream of the tokens it builds. The `start-pos` and `end-pos` variables refer to the locations at the start of the match and the end of the match respectively. A lexer defined with `lexer-src-pos` instead of `lexer` automatically packages the action's return value with the matched text's starting and ending positions. This relieves the programmer from having to manage location information in each action.[3]

4.3 Code Generation

Most lexer generators first convert the regexps to a nondeterministic finite automaton (NFA) using Thompson's construction [18] and then use the subset construction to build a DFA, or they combine these two steps into one [1, section 3.9]. The naive method of handling complement in the traditional approach applies Thompson's construction to build an NFA recursively over the regexp. When encountering a complement operator, the subset construction is applied to convert the in-progress NFA to a DFA which is then easily complemented and converted back to an NFA. Thompson's construction then continues. We know of no elegant method for handling complement in the traditional approach. However, the DMS system [2] uses the NFA to DFA and back method of complement and reports practical results.[4]

The PLT lexer generator builds a DFA from its component regular expressions following the derivative based method of Brzozowski [4]. The derivative approach builds a DFA directly from the regexps, and handles complement and intersection exactly as it handles union.

Given a regular expression r, the *derivative* of r with respect to a character c, $D_c(r)$, is $\{s \mid r \text{ matches } cs\}$. The derivative of a regexp can be given by another regexp, and Brzozowski gives a simple recursive function that computes it. The DFA's states are represented by the regexps obtained by repeatedly taking derivatives with respect to each character. If $D_c(r) = r'$, then the state r has a transition on character c to state r'. Given r and its derivative r', the lexer generator needs to determine whether a state equivalent to r' already exists in the DFA. Brzozowski shows that when comparing regexps by equality of the languages they denote, the iterated derivative procedure constructs the minimal DFA. Because of the complexity of deciding regular language equality, he also shows that the process will terminate with a (not necessarily minimal) DFA if regexps are compared structurally, as long as those that differ only by associativity, commutativity and idempotence of union are considered equal.

[3]The `return-without-pos` variable lets src-pos lexers invoke other src-pos lexers without accumulating multiple layers of source positioning.

[4]Michael Mehlich, personal communication

A few enhancements render Brzozowski's approach practical. First, the regexp constructors use a cache to ensure that equal regexps are not constructed multiple times. This allows the lexer generator to use `eq?` to compare expressions during the DFA construction. Next, the constructors assign a unique number to each regexp, allowing the sub-expressions of a union operation to be kept in a canonical ordering. This ordering, along with some other simplifications performed by the constructors, guarantees that the lexer generator identifies enough regexps together that the DFA building process terminates. In fact, we try to identify as many regexps together as we can (such as by canceling double complements and so on) to create a smaller DFA.

With modern large characters sets, we cannot efficiently take the derivative of a regexp with respect to each character. Instead, the lexer generator searches through the regexp to find sets of characters that produce the same derivative. It then only needs to take one derivative for each of these sets. Traditional lexer generators compute sets of equivalent characters for the original regexp. Our derivative approach differs in that the set is computed for each regexp encountered, and the computation only needs to consult parts of the regexp that the derivative computation could inspect.

Owens added the derivative-based lexer generator recently. Previously, the lexer generator used a direct regexp to DFA algorithm [1, section 3.9] (optimized to treat character sets as single positions in the regexp). Both algorithms perform similarly per DFA state, but the derivative-based algorithm is a much better candidate for elegant implementation in Scheme and may tend to generate smaller DFAs. On a lexer for Java, both algorithms produced (without minimization) DFAs of similar sizes in similar times. On a lexer for Scheme, the Brzozowski algorithm produced a DFA about $\frac{2}{5}$ the size (464 states vs. 1191) with a corresponding time difference.

4.4 Summary

Embedding the lexer generator into Scheme places the action expressions naturally into their containing program. The embedding relies on hygienic macro expansion. To support convenient complement and intersections, we moved the lexer generator from a traditional algorithm to one based on Brzozowski's derivative. Even though the derivative method is not uniquely applicable to Scheme, we found it much more pleasant to implement in Scheme than our previous DFA generation algorithm.

5 Parser Generators

A parser is built from a CFG and consumes the tokens supplied by the lexer. It matches the token stream against the CFG and evaluates the corresponding attributes, often producing an AST.

5.1 Grammars

A CFG consists of a series of definitions of *non-terminal* symbols. Each definition contains the non-terminal's name and a sequence of terminal and non-terminal symbols. Uses of non-terminals on the right of a definition refer to the non-terminal with the same name on the left of a definition. A *terminal* symbol represents an element of the token stream processed by the parser.

A parser cannot, in general, efficiently parse according to an arbitrary CFG. The bottom-up parsers generated by YACC use a lookahead function that allows them to make local decisions during parsing and thereby parse in linear time. Lookahead computation has

ambiguous results for some grammars (even unambiguous ones), but the LALR(1) lookahead YACC uses can handle grammars for most common situations. Precedence declarations allow the parser to work around some ambiguities. Both the PLT and GT tools follow YACC and use LALR(1) lookahead with precedences.

5.2 Tokens

A parser is almost completely parametric with respect to tokens and their associated values. It pushes them onto the value stack, pops them off it, and passes them to the semantic actions without inspecting them. The parser *only* examines a token when it selects shift/reduce/accept actions based on the tokens in the input stream's lookahead buffer. This is a control dependency on the token representation because the parser must perform a conditional branch that depends on the token it sees.

Nevertheless, most parser generators, including the PLT system, enforce a specific token representation. The PLT system abstracts the representation so that, were it to change, existing lexer/parser combinations would be unaffected. The GT system allows the token representation to be specified on a parser-by-parser basis.

5.2.1 Tokens in GT

The GT parser tool is parameterized over token branch computation; it has no knowledge of the token representation otherwise. The GT parser macro takes the name of a `token-case` macro along with the CFG specification. The parser generator uses the `token-case` macro in the multi-way branch forms it produces:

```
(token-case token-exp
  ((token ...) body ...)
  ...
  (else body ...))
```

The Scheme expression *token-exp* evaluates to a token value, and the *token* elements are the token identifiers declared with the CFG. Scheme's macro hygiene ensures that the identifiers declared in CFG token declarations and the keys recognized by the `token-case` macro interface properly.

The `token-case` macro has a free hand in implementing the primitive token-branch computation. It can produce a type test, if tokens are Scheme values such as integers, symbols, and booleans; extract some form of an integer token-class code from a record structure, if tokens are records; or emit an immediate jump table, if tokens are drawn from a dense space such as the ASCII character set.

The `token-case` branch compiler parameterizes the CFG to Scheme compiler. This ability to factor compilers into components that can be passed around and dropped into place is unique to Scheme. Note, also, that this mechanism has nothing to do with core Scheme *per se*. It relies only on the macro technology which we could use with C or SML, given suitable s-expression encodings.

5.2.2 Tokens in PLT

The PLT parser generator sets a representation for tokens. A token is either a symbol or a token structure containing a symbol and a value, but this representation remains hidden unless the programmer explicitly queries it. A programmer declares, outside of any parser or lexer, the set of valid tokens using the following forms for tokens with values and without, respectively.

```
(define-tokens group-name (token-name ...))
(define-empty-tokens group-name
  (token-name ...))
```

A parser imports these tokens by referencing the group names in its `tokens` argument. The parser generator statically checks that every grammar symbol on the right of a production appears in either an imported token definition or on the left of a production (essentially a non-terminal definition). DrScheme reports violations in terms of the CFG, as discussed in Section 6.2.

The token-declaration forms additionally provide bindings for token-creation functions that help ensure that the lexer creates token records in conjunction with the parser's expectations. For example, `(token-x)` creates an empty token named `x`, and `(token-y val)` creates a non-empty token named `y`. Thus the single external point of token declaration keeps the token space synchronized between multiple lexers and parsers.

5.3 Parser Configuration

A parser specification contains, in addition to a CFG, directives that control the construction of the parser at an operational level. For example, precedence declarations, in the GT `tokens` and PLT `precs` forms, resolve ambiguities in the CFG and in the lookahead computation.

The PLT `start` form declares the non-terminal at the root of the parse tree. When multiple `start` non-terminals appear, the parser generator macro expands into a list containing one parser per `start` non-terminal. Multiple `start` symbols easily allow related parsers to share grammar specifications. (Most other parser generators do not directly support multiple `start` symbols and instead require a trick, such as having each intended `start` symbol derive from the real `start` symbol with a leading dummy terminal. The lexer produces a dummy terminal to select the desired `start` symbol [17, section 10].)

The PLT system `end` form specifies a set of distinguished tokens, one of which must follow a valid parse. Often one of these tokens represents the end of the input stream. (Other parser generators commonly take this approach.) In contrast, GT's `accept-lookaheads` clause supports k-token specifications for parse ends. Thus nothing in GT's CFG language is specifically LALR(1); it could just as easily be used to define an LR(k) grammar, for $k > 1$. Although the current tools only process LALR(1) grammars, the CFG language itself allows other uses.

GT's CFG language makes provision for the end-of-stream (eos) as a primitive syntactic item distinct from the token space. An `accept-lookaheads` specification references eos with the `#f` literal, distinguishing the concept of end-of-stream (absence of a token; a condition of the stream itself) from the set of token values. This was part of clearly factoring the stream representation (*e.g.*, a list, a vector, an imperative I/O channel) from the token representation (*e.g.*, a record, a character, a symbol) and ensures that the token space is not responsible for encoding a special end-of-stream value.

Figure 6 The PLT parser language

```
parser ::= (parser-clause ...)

parser-clause ::= (start nterm ...)                  ; Starting non-terminals
                | (end term ...)                     ; Must follow parse only
                | (tokens token-group ...)           ; Declare tokens
                | (error scheme-exp)                 ; Called before error correction
                | (grammar (nterm rhs ...)  ...)     ; Defines the grammar
                | (src-pos)                          ; Optional: Automatic source locationing
                | (precs (prec term ...)  ...)       ; Optional: Declare precedences
                | (debug filename)                   ; Optional: print parse table & stack on error
                | (yacc-output filename)             ; Optional: Output the grammar in YACC format
                | (suppress)                         ; Optional: Do not print conflict warnings

rhs      ::= ((gsym ...) [term] action)   ; Production with optional precedence tag
prec     ::= left | right | nonassoc      ; Associativity/precedence declarator
gsym     ::= term | nterm                 ; Grammar symbol
action   ::= scheme-exp                   ; Semantic action
filename ::= string
term, nterm, token-group ::= identifier
```

5.4 Attribute Computation

Each production in the grammar has an associated expression that computes the attribute value of the parse-tree node corresponding to the production's left-hand non-terminal. This expression can use the attribute values of the children nodes, which correspond to the grammar symbols on the production's right side. In YACC, the variable `$n` refers to the value of the nth grammar symbol.

The PLT system non-hygenically introduces `$n` bindings in the attribute computations. Tokens defined with `define-empty-tokens` have no semantic values, so the parser form does not bind the corresponding `$n` variables in the semantic actions. The parser generator thereby ensures that a reference to a `$n` variable either contains a value or triggers an error.

Instead of requiring the `$n` convention, the GT design places no restrictions on the variables bound to attribute values. For example, the subtraction production from a simple calculator language,

```
(non-term exp
  ...
  (=> ((left exp) - (right exp))  (- left right))
  ...)
```

hygienically introduces the `left` and `right` bindings referenced from the attribute computation, `(- left right)`. A grammar symbol without enclosing parentheses, such as `-`, specifies no binding, indicating to downstream tools that the token's semantic value may be elided from the value stack when it is shifted by the parser. (Essentially, the empty-token specification shows up in the CFG specification.)

As a convenience syntax, if the variable is left unspecified, as in

```
(=> ((exp) - (exp))  (- $1 $3))
```

then the `$n` convention is used. This unhygienic bit of syntactic sugar is convenient for hand-written parsers, while the explicit-binding form provides complete control over variable binding for hygienic macros that generate CFG forms.

For further convenience, the `implicit-variable-prefix` declaration can override the `$` prefix. Thus, a handwritten parser can arrange to use implicitly-bound variables of the form `val-1`, `val-2`, ..., with the declaration

```
(implicit-variable-prefix val-)
```

The `$n` notation is unavailable in the Bigloo parser generator. Instead, the grammar symbol's name is bound to its value in the action. Because the same grammar symbol could appear more than once, the programmer can choose the name by appending it to the grammar symbol's name with the `@` character in-between. The Bigloo design provides more naming naming control than the PLT system, but no more control over hygiene. Additionally, it can lead to confusion if grammar symbols or attribute bindings already contain `@`.

5.5 Code Generation

Although the PLT and GT parser generators are based on YACC's design, both use a syntactic embedding of the parser specification into Scheme, much as PLT's lexer generator does. In the PLT system, a programmer writes a parser by placing a specification written in the language shown in Figure 6 inside of a `(parser ...)` form. The `(parser ...)` form compiles the grammar into a parse table using LALR(1) lookahead and supplies an interpreter for the table. These two elements are packaged together into a parser function. The GT parser system uses the language in Figure 7 for its CFG specifications. As in the PLT system, a CFG specification is placed inside of a macro that compiles the CFG form into the target language. However, the GT design provides a much wider range of implementation options than the PLT and other systems.

The GT system factors the parser tool-chain into multiple languages. The programmer writes a parser using the CFG language and the parser generator compiles it to a Scheme implementation in three steps. It transforms the CFG into a TLI (for "target-language independent") specification which it then expands to an equivalent parser in a push-down automata (PDA) language which it finally compiles into Scheme. The continuation-passing-style (CPS) macro `(cfg->pda` *cfg form* `...)` packages up the LALR com-

Figure 7 The GT CFG language

```
cfg ::= (clause ...)

clause ::= (tokens token-decl ...)                          ; Declare tokens and precedence tags
         | (non-term nterm rhs ...)                         ; Declare a non-terminal
         | (accept-lookaheads lookahead ...)                ; Must come after parse
         | (error-symbol ident [semantic-value-proc])       ; Error-repair machinery
         | (no-skip token ....)                             ; Error-repair machinery
         | (implicit-variable-prefix ident)                 ; Defaults to $
         | (allowed-shift/reduce-conflicts integer-or-false)

token-decl ::= token
             | (non token ...)      ; Non-associative tokens
             | (right token ...)    ; Right-associative tokens
             | (left token ...)     ; Left-associative tokens

rhs ::= (=> [token] (elt ...) action)    ; Production w/optional precedence tag

elt ::= symbol           ; Symbol w/unused semantic value
      | (var symbol)     ; Symbol binding semantic value to var
      | (symbol)         ; Symbol w/implicitly bound semantic value

lookahead ::= (token ...[#f])    ; #f marks end-of-stream.
action ::= scheme-exp
symbol, nterm, token, var ::= ident
```

piler machinery and performs the first two steps. It expands to (*form ... pda*), where *pda* is the PDA program compiled from the original *cfg* form. The macro `pda-parser/imperative-io` takes a PDA program, along with the `token-case` macro other relevant forms specifying the input-stream interface, and expands it into a complete Scheme parser. An alternate macro, `pda-parser/pure-io` maps a PDA program to Scheme code using a functional-stream model; it is intended for parsing characters from a string, or items from a list. The main parser macro simply composes the `cfg->pda` macro with one of the PDA-to-Scheme macros to get a Scheme parser; this is a three-line macro.

Exporting the PDA language lets the system keep the token-stream mechanism abstract throughout the CFG-to-PDA transformation. The two PDA-to-Scheme macros each provide a distinct form of token-stream machinery to instantiate the abstraction. In contrast, the PLT system fixes the token-stream representation as a function of no arguments that returns a token. Successive calls to the function should return successive tokens from the stream.

5.5.1 The TLI language

GT system was designed to be target-language neutral. That is, to specify a parser in C instead of in Scheme using the CFG language, we would only need an s-expression concrete grammar for C in which to write the semantic actions. This means that the CFG-processing tools for the GT system are also independent of the target language and the language used for the semantic actions. Note that Scheme creeps out of the semantic actions and into the rest of the grammar language in only one place: the variable-binding elements of production right-hand sides. These variables (such as the `left` and `right` variables bound in the above example) are Scheme constructs.

To excise this Scheme dependency, the GT system defines a slightly lower-level language than the CFG language defined in Figure 7. The lower-level language (called the TLI language) is identical to the main CFG language, except that (1) the `implicit-variable-prefix` clause is removed (having done its duty during the CFG-to-TLI expansion), and (2) variable binding is moved from the production rhs to the semantic-action expression. In the TLI language, the example above is rewritten to

```
(=> ((exp) - (exp))                          ; Grammar
    (lambda (left right) (- left right)))    ; Scheme
```

As in the the main CFG language, parentheses mark grammar symbols whose semantic values are to be provided to the semantic action. The TLI language is *completely* independent of the target language, except for the semantic actions. In particular, TLI has nothing to do with Scheme at all. This means that the CFG can be compiled to its push-down automaton (PDA) with complete indifference to the semantic actions. They pass through the LALR transformer unreferenced and unneeded, to appear in its result. Because the TLI language retains the information about which semantic values in a production are needed by the semantic action, optimizations can be performed on the parser in a target-language independent manner, as we will see below.

5.5.2 The PDA language

A PDA program (see Figure 8) is primarily a set of *states*, where each state is a collection of shift, reduce, accept and goto *actions*. Shift, reduce and accept actions are all guarded by *token lookahead* specifications that describe what the state of the token stream must be in order for the guarded action to fire. A non-terminal symbol guards a goto action. Reduce actions fire named *rules*, which are declared by `rule` clauses; a reduction pops semantic values off the value stack and uses its associated semantic action to compute the replacement value.

The PDA design contains several notable elements. The lookahead syntax allows for k-token lookahead, for $k = 0, 1$ and greater, so the PDA language supports the definition of LR(k) parsers (al-

Figure 8 The PDA language

```
pda ::= (pda-clause ...)

pda-clause ::= (comment form ...)                                  ; Ignored
             | (tokens token ...)                                  ; Declare tokens
             | (state state-name action ...)
             | (rule rule-name non-term bindings semantic-action)
             | (error-symbol ident [semantic-value-proc])          ; Error-repair machinery
             | (no-skip token ...)                                 ; Error-repair machinery

action ::= (comment form ...)              ; Ignored
         | (shift lookahead state-name)    ; Shift, reduce & accept
         | (reduce lookahead rule-name)    ;    actions all guarded
         | (accept lookahead)              ;    by token-lookaheads.
         | (goto non-term state-name)      ; Goto action guarded by non-terminal
         | (error-shift ident state-name)  ; Error-repair machinery

lookahead ::= (token ...[#f])   ; #f marks end-of-stream
bindings  ::= (boolean ...)     ; #f marks a value not passed to semantic action
state-name, rule-name ::= ident
token, non-term ::= ident
```

though our tools only handle $k \leq 1$). As action-selection is order-dependent, the zero-token lookahead `()` is useful as a default guard.

Also notable, the *bindings* element of the rule form is a list of boolean literals, whose length determines how many semantic values are popped off the value stack. Only values tagged with a `#t` are passed to the semantic action; values tagged with a `#f` are discarded. As an example, the reduction rule

```
;;;  ifexp ::= if <exp> then <stmt> else <stmt> fi
(rule r7 ifexp (#t   #t   #f   #t   #f   #t   #t)
  (lambda (iftok exp stmt1 stmt2 fitok)
    (make-ifexp exp stmt1 stmt2
       (token:leftpos  iftok)    ;Position-tracking
       (token:rightpos fitok)))) ;machinery
```

specifies a rule that will pop seven values off the stack, but only pass five of them to the semantic action. Thus, the semantic action is a Scheme procedure that takes only five arguments, not seven.

The bindings element allows a PDA optimizer, via static analysis, to eliminate pushes of semantic values that will ultimately be discarded by their consuming reductions—in effect, useless-value elimination at the PDA level. The *bindings* form specifies the local data dependencies of the semantic actions. This key design point allows data-flow analysis of the PDA program *without requiring any understanding of the language used to express the semantic action*, which in turn supports strong linguistic factoring. The semantic action s-expression could encode a C statement, or an SML expression just as easily as a Scheme expression; a PDA optimizer can analyse and transform a PDA program with complete indifference.

A great allure of PDA computation is its sub-Turing strength, which means that we have a much easier time analyzing PDA programs than those written in a Turing-equivalent language. The moral might be: always use a tool small enough for the job. We have designed and are currently implementing a lower-level PDA0 language, which allows source-to-source optimizations such as non-terminal folding, control- and data-flow analysis, and dead-state elision. This has the potential to make very lightweight parsing practical, *i.e.*, parsers that parse all the way down to individual characters, yet still assemble tokens at lexer speeds. Again, this can all be provided completely independent of the eventual target language by defining CPS macros that work strictly at the PDA0 level.

Factoring out the PDA as a distinct language also supports multiple producers as well as multiple consumers of PDA forms. One could implement SLR, canonical LR and other CFG processors to target the same language, and share common back end.

5.6 Summary

Although the PLT and GT parser generators follow the general design of YACC, both systems syntactically embed parser specifications in Scheme. The embedding benefits the PLT parser generator in the same way it benefits the PLT lexer generator, whereas the GT system takes advantage of the syntactic embedding to maximize flexibility. In GT, the token structure is specified on a per-parser basis through a `token-case` macro, avoiding any commitment to a particular lexer/parser interface. (The PLT token strategy could be implemented as a `token-case` macro.) Furthermore, the GT system provides complete freedom over naming in the grammar and attributes, without compromising hygiene. We think GT's attribute naming system is superior to other Scheme parser generators, including Bigloo's and PLT Scheme's. By using a language tower, the GT system can isolate details of one level from the others. This allows, for example, easily switching between multiple PDA implementations and token stream representations with the same CFG. The separation of the token and end-of-stream representations supports the use of different kinds of token-stream representations.

6 Taking Full Advantage of Syntax

As we have seen, syntactic embeddings of lexer and parser specifications allow the lexer and parser generator to perform the translation to DFA and PDA at compile time. The syntactic approach also supports the debugging of parser specifications and lets the program development environment operate on them.

6.1 Debugging Parsers

Static debugging of an LR parser has two components: detecting malformed grammars and semantic actions, and detecting grammars that do not conform to the requirements of the parsing methodology in use. The PLT system helps programmers with the former kinds of error using the techniques mentioned in Section 5.2 and Section 6.2. The GT system's multi-level design gives programmers an elegant way of approaching the latter kinds of problems.

Most parsing methodologies (including LL(k), LR(k), and LALR) cannot handle all unambiguous CFGs, and each builds ambiguous PDAs on some class of unambiguous CFGs. Analyzing and fixing these grammars necessarily requires an examination of the item sets associated with the conflicted states of the broken PDA—they must be debugged *at the PDA level*. In most systems, including YACC and the PLT system, these errors are debugged by printing out and then carefully studying a report of the grammar's ambiguous characteristic finite-state automaton (CFSA), which is essentially the program defining the PDA.

An ambiguous PDA has multiple shift/reduce/accept transitions guarded by the same lookahead, so the check for bad grammars occurs statically in the PDA-to-Scheme macro. Because the GT parser system factors the parser tool chain into multiple language levels, the report machinery comes for free: the PDA program is the report. Since the LALR compiler is exported as a CPS macro, using `quote` for the syntactic continuation shifts from language-level to data structure. That is, this Scheme form `(cfg->pda` *cfg* `quote)` expands to `(quote` *pda*`)` so the the following expression produces an error report.

```
(pretty-print (cfg->pda cfg quote))
```

The PDA language includes a `comment` clause for the LALR compiler to record the item-set information for each state. This information is critical for human understanding of the PDA. An example of a state generated by `cfg->pda` is

```
(state s15
  (comment (items (=> exp (exp () divide exp))
                  (=> exp (exp () times exp))
                  (=> exp (exp minus exp ()))
                  (=> exp (exp () minus exp))
                  (=> exp (exp () plus exp))))
  (reduce (r-paren) r11)
  (reduce (semicolon) r11)
  (comment (reduce (times) r11))
  (shift (times) s11)
  (comment (reduce (divide) r11))
  (shift (divide) s12)
  (comment (reduce (plus) r11))
  (shift (plus) s13)
  (comment (reduce (minus) r11))
  (shift (minus) s14)
  (reduce (#f) r11))
```

A comment clause lists the kernel set of the state's items. The item comments are grammar productions with `()` allowed on the right-hand sides to mark the item cursor. (We did not use the traditional dot marker '.' for obvious reasons.) The LALR compiler comments out ambiguous actions that are resolved by precedence, associativity, or the allowed-conflict-count declaration. State s15 has four of these. Had one of them not been resolved by the LALR macro, the resulting PDA would be ambiguous, causing the PDA macro to report a static error that the programmer would have to debug.

The GT tools also have small touches to help the programmer focus in on the problem states. The LALR compiler leaves a simple report in a comment at the top of the PDA form listing the names of all conflicted states, *e.g.*,

```
(comment (conflict-states s41 s63 s87))
```

The GT tools also provide a PDA analyser that filters a PDA and produces a reduced PDA that containing only the ambiguous states of the original program. Because we can so trivially render the PDA as a Scheme s-expression, it is easy to comb through a PDA or otherwise interactively manipulate it using the usual suite of Scheme list-processing functions such as `filter`, `fold`, `map`, `any` and so forth—a luxury not afforded to YACC programmers.

Placing PDA static-error detection in the PDA tools, where it belongs, has another benefit. Since the LALR compiler will happily produce an ambiguous PDA, we could produce a generalized LR (GLR) parser simply by implementing a nondeterministic PDA as a distinct macro from the current PDA-to-Scheme one. It would compose with the current `cfg->pda` macro, handling ambiguous grammars without complaint allowing reuse of the complex LALR tool with no changes.

6.2 Little Languages and PDEs

The DrScheme program development environment has several features that display feedback directly on a program's source. Specifically, DrScheme highlights expressions that have caused either a compile-time or run-time error, and the Check Syntax tool draws arrows between binding and bound variables. Check Syntax inspects fully expanded Scheme source code to determine arrow placements. The action and attribute expressions inside the PLT `lexer` and `parser` forms appear directly in the expanded code with their lexical scoping and source location information intact, so that Check Syntax can draw arrows, and DrScheme can highlight errors as demonstrated in Figures 1 and 2 in Section 2.2.

The `lexer` and `parser` forms expand into DFA and parse tables, leaving out the source regular expression and CFG specifications. Thus, DrScheme requires extra information to fully support these forms. Run-time error highlighting is not an issue, because the regexp or grammar itself cannot cause a runtime error. The `lexer` and `parser` forms directly signal compile-time errors (*e.g.*, for an unbound regexp operator or terminal), including the source location of the error, to DrScheme. As they parse the input regexp or grammar expression, each sub-expression (as a syntax object) contains its source location, so they can conveniently signal such errors.

To inform Check Syntax of dependencies in the grammar, the `parser` form emits a dummy `let` form as dead code, along with the parse table and actions. The `let` includes a binding for each non-terminal and token definition, and its body uses each grammar symbol that occurs on the right of a production. The `let` introduces a new scope for all of the non-terminals and tokens, ensuring that they do not interfere with outside identifiers of the same name. The `parser` form generates the following `let` for the example in Figure 9.

```
(let ((exp void)
      (NUM void)
      (– void)
      (EOF void))
  (void NUM exp – exp))
```

Figure 9 Locating the uses of a token and a non-terminal

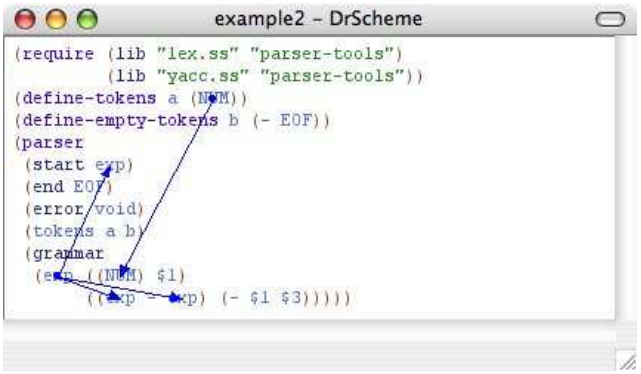
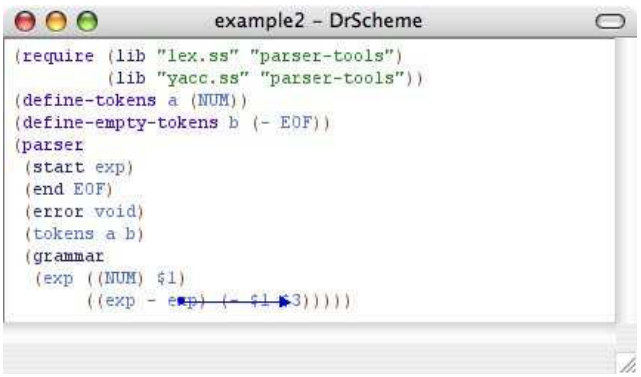


Figure 10 Correlating an action with the grammar



We use a different approach for the situation shown in Figure 10. The parser generator wraps the action with a `lambda` that binds the `$3`. To cause Check Syntax to draw an arrow, the `lambda`'s `$3` parameter uses the source location of the referent grammar symbol. With a GT-style hygienic naming option, we would use the identifier supplied with the grammar symbol in the `lambda` instead, and Check Syntax could then draw the arrow appropriately to the binder. Furthermore, α-renaming could be used to change the name. This illustrates that hygienic macros interact more naturally with programming tools, and not just with other macros.

Like any compiler, a macro that processes an embedded language must respect that language's dynamic semantics by generating code that correctly executes the given program. Also like any compiler, the macro must implement the language's static semantics. It can do this by performing the requisite static checking itself, as in the SRE type system and the PLT `parser` form's check for undefined grammar symbols, or it can arrange for statically invalid source programs to generate statically invalid target programs. In this case, the macro effectively re-uses the target language's static checking. This is how the `parser` form handles unbound `$n` identifiers, by letting Scheme's free variable detection catch them. Even for the static properties checked directly by the macro, it might need to emit annotations (such as the `let` mentioned above) to preserve static information for tools like Check Syntax.

7 References

[1] A. A. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *International Conference on Software Engineering*, 2004.

[3] D. Boucher. A portable and efficient LALR(1) parser generator for Scheme. `http://www.iro.umontreal.ca/~boucherd/Lalr/documentation/lalr.html`.

[4] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[5] F. DeRemer and T. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649, 1982.

[6] D. Dubé. SILex. `http://www.iro.umontreal.ca/~dube/`.

[7] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[8] M. Flatt. *PLT MzScheme: Language Manual*, 2004. `http://download.plt-scheme.org/doc/mzscheme/`.

[9] D. P. Friedman, M. Wand, and C. P. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2001.

[10] M. Johnson. `http://cog.brown.edu:16080/~mj/Software.htm`.

[11] M. Serrano. *Bigloo: A "practical Scheme compiler"*, 2004. `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.

[12] O. Shivers. A scheme shell. *Higher-order and Symbolic Computation*. to appear.

[13] O. Shivers. A universal scripting framework, or lambda: the ultimate "little language". In *Concurrency and Parallelism, Programming, Networking, and Security*, volume 1179 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 1996.

[14] O. Shivers. The SRE regular-expression notation. `http://www.cs.gatech.edu/~shivers/sre.txt`, 1998.

[15] O. Shivers and B. Carlstrom. The scsh manual. `ftp://www-swiss.ai.mit.edu/pub/su/scsh/scsh-manual.ps`.

[16] M. Sperber and P. Thiemann. Generation of LR parsers by partial evaluation. *ACM Trans. Program. Lang. Syst.*, 22(2):224–264, 2000.

[17] D. R. Tarditi and A. W. Appel. *ML-Yacc User's Manual: Version 2.4*, 2000. `http://www.smlnj.org/doc/ML-Yacc/`.

[18] K. Thompson. Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

Compiling Java to PLT Scheme

Kathryn E. Gray Matthew Flatt
Univeristy of Utah

Abstract

Our experimental compiler translates Java to PLT Scheme; it enables the use of Java libraries within Scheme programs, and it makes our Scheme programming tools available when programming with Java. With our system, a programmer can extend and use classes from either language, and Java programmers can employ other Scheme data by placing it in a class using the Java native interface.

PLT Scheme's class-system, implemented with macros, provides a natural target for Java classes, which facilitates interoperability between the two languages, and PLT Scheme's `module` maintains Java security restrictions in Scheme programs. Additionally, `module`'s restrictions provide a deeper understanding of a Java compilation unit and make Java's implicit compilation units explicit.

1 Why Compile Java to Scheme?

Scheme implementations that compile to Java (or JVM bytecode) benefit from the extensive infrastructure available for Java programs, including optimizing just-in-time compilers, JVM debugging tools, and an impressive roster of Java-based libraries. For PLT Scheme, we have inverted the equation, compiling Java to Scheme. We thus obtain a Java implementation with access to PLT Scheme's libraries and facilities—especially the DrScheme environment and its teaching modes [5], which is the primary motivation for our effort [9].

By compiling Java to Scheme, we also gain access to the many libraries implemented in Java, as long as we can bridge the gap between Java and Scheme. In many ways, the translation is the same for Java-to-Scheme compilation as it is for Scheme-to-Java, but the trade-offs are somewhat different. In particular, libraries that contain native calls are no problem for Scheme-to-Java compilation, but Java-to-Scheme must provide special support for native methods. In contrast, a Scheme compilation model with expressive macros accommodates Java code more easily than Java's model of compilation accommodates Scheme.

Our Java-to-Scheme compiler remains a work in progress. Even so, we have gained experience that may be useful to future implementors of Java-to-Scheme compilers.

In the long run, we expect that many useful Java libraries will fit into our implementation, as PLT Scheme provides a significant set of libraries of its own. For example, we expect that the subset of AWT used by Swing can be mapped onto PLT Scheme's GUI primitives, thus enabling Swing-based applications to run in PLT Scheme. In other words, we believe that many Java libraries can be made to run by re-implementing certain "native" Java methods in PLT's native language, Scheme.

In this report, we describe

- a strategy for compiling Java classes to PLT Scheme, which exploits a macro-implemented extension of Scheme for object-oriented programming;
- the interaction of the strategy with PLT Scheme's module system;
- how we define the translation of run-time values between Java and Scheme; and
- open problems that we have not yet addressed.

Before introducing our compilation strategy, we begin with a description of two libraries that we would like to embed in Scheme, and the strategy of doing so. One is relatively easy to support, and the other is more difficult.

2 Java Libraries

Kyle Siegrist's probability and statistics library (PSOL) [16] provides various mathematical procedures. The library also deploys a graphical interface to experiment with the procedures, but the interface is not necessary to use the library's primary functionality.

Terence Parr's ANTLR [13] provides parsing technology that permits grammar inheritance.

These two libraries are representative of the kinds of code and dependencies found in non-graphical Java libraries. We estimate that roughly 10 to 15% of Java libraries have requirements similar to PSOL, while 30 to 40% are similar to ANTLR. The remaining libraries tend to depend on graphic capabilities.

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

Naturally, both packages rely on the basic components of Java's class system, including classes and interfaces, overloading, and static (as well as instance) members, as well as loops and arrays. Both libraries also depend on some of Java's core libraries, including `Object`, `String`, and `Throwable`.

PSOL depends on Java's `Math` library and numeric wrapper classes. The latter provides the ability to use primitive values (such as `1.2`) as objects, as well as functionality over numbers. These libraries rely on native methods with existing counterparts in PLT Scheme.

ANTLR requires several language features that we do not yet support: nested classes (commonly known as inner classes), the `switch` statement, and reflection. The first two, nested classes and `switch`, simply have not been implemented yet in our system, but there are no technical challenges. Reflection is more difficult than the other two, though, and we discuss this problem in Section 5.5.2. ANTLR further relies on utility libraries and `IO`, which in turn rely on nested classes and reflection. Several `IO` classes rely on native methods that have counterparts in PLT Scheme.

Neither PSOL nor ANTLR immediately ran in our current implementation. In the case of PSOL, we easily implemented the relevant `Math` methods and wrapper classes, so that PSOL now runs in non-graphical mode. We expect that implementing the `IO` methods needed for ANTLR will be similarly easy, but reflection support is a much larger obstacle. Support for graphical PSOL is completely out of reach in the short term.

3 Classes and Objects in PLT Scheme

PLT Scheme was designed from the start to support GUI programming with class-based, object-oriented constructs. Originally, the class implementation was built into the interpreter's core, and each object was implemented as a record of closures. Our current system is more Java-like, in that an object is a record of fields plus a per-class method table, and it is implemented outside the core by a collection of (an extended) `syntax-case` macros.

3.1 Class Constructs

A class is created with the keyword `class`, and the resulting form is much like Java's. It creates a new class given a superclass and a collection of method overrides, new methods, and new fields. Syntactically, `class` consists of an expression for the superclass followed by a sequence of declarations.

The following is a partial grammar for `class` clauses:

```
expr        =  ...
            |  (class super-expr clause ...)
clause      =  (field init-decl ...)
            |  expr
            |  (init init-decl ...)
            |  (public name ...)
            |  (override name ...)
            |  (private name ...)
            |  (define name method)
            |  (inherit name ...)
init-decl   =  name
            |  (name init-expr)
method      =  (lambda formals expr0 expr ...)
```

Instead of a constructor, a class contains `field` declarations and other *expr*s to evaluate each time the class is instantiated. An `init` introduces a keyword-based initialization argument (possibly with a default value) to be supplied when the class is instantiated. The `public` form names the new public methods defined in the class, `override` names the overriding methods defined in the class, `private` names private methods. Each method definition looks like a function definition using `define` and a name declared as `public`, `override`, or `private`. Macros such as `define/public` (not shown in the grammar) combine the declaration and definition into a single form.

The `inherit` form names methods to be inherited from the superclass. Inherited methods must be declared explicitly because classes are first-class values in PLT Scheme, and a class's superclass is designated by an arbitrary expression. The advantage of first-class classes is that a mixin can be implemented by placing a `class` form inside a `lambda` [6]. Obviously, Java code contains no mixin declarations, but as we note later in Section 7.2, mixins provide a convenient solution to certain interoperability problems.

The built-in class `object%` serves as the root of the class hierarchy, and by convention, `%` is used at the end of an identifier that is bound to a class. As in Java, the `class` system supports only single-inheritance. A class explicitly invokes the body expressions of its superclass using `super-new`.

Within a `class` body, fields and methods of the class can be accessed directly, and fields can be modified using `set!`. Initialization arguments can be accessed only from field initializers and other expressions, and not in method bodies. For example, a stack class can be implemented as follows:

```
(define stack%
  (class object%
    (public push pop)
    (init starting-item)
    (field (content (list starting-item)))
    (define push
      (lambda (v)
        (set! content (cons v content))))
    (define pop
      (lambda ()
        (let ((v (car content)))
          (set! content (cdr content))
          v)))
    (super-new)))
```

The `new` form instantiates a class. Its syntax is

```
(new class-expr (init-name expr) ...)
     ; ⇒ an object
```

where each *init-name* corresponds to an `init` declaration in the class produced by *class-expr*. For example, a stack instance (that initially contains a `5`) is created as follows:

```
(define a-stack (new stack% (starting-item 5)))
```

The `send` form invokes a method of an instance,

```
(send obj-expr method-name arg-expr ...)
      ; ⇒ method result
```

where *method-name* corresponds to a `public` method declaration in *obj-expr*'s class (or one of its super classes). For example, `send` is used to push and pop items of *a-stack*:

```
(send a-stack push 17)
(send a-stack pop) ;  ⇒ 5
```

Each execution of `send` involves a hash-table lookup for *method-name*; to avoid this overhead for a specific class, a programmer can obtain a generic function using `generic` and apply it with `send-generic`:

```
(generic class-expr method-name); ⇒ a generic
(send-generic obj-expr generic-expr
      arg-expr ...) ;  ⇒ method result
```

In accessing fields, the forms `class-field-accessor` and `class-field-mutator` produce procedures that take an instance of the given class and get or set its field.

```
(class-field-accessor class-expr field-name)
(class-field-mutator class-expr field-name)
```

where *field-name* corresponds to a `field` declaration in the class.

By default, a `field`, `init`, or method name has global scope, in the same sense as a symbol. By using global scope for member names, classes can be easily passed among modules and procedures (for mixins or other purposes).

A name can be declared to have lexical scope using `define-local-member-name`:

```
(define-local-member-name name ...)
```

When a *name* is so declared and used with `init`, `field`, or `public`, then it is only accessible through `override`, `inherit`, `new`, `send`, `generic`, `class-field-accessor`, and `class-field-mutator` in the scope of the declaration. At PLT Scheme's module level, local member names can be imported and exported, just like macros. At run-time, the values produced by `generic`, `class-field-accessor`, and `class-field-mutator` can be used to communicate a method or field to arbitrary code.

For example, we can use `define-local-member-name` to make the *content* field private[1] to the scope of the *stack%* declaration:

```
(define-local-member-name content)
(define stack% (class ...))
(define stack-content
  (class-field-accessor stack% content))
(define (empty-stack? s)
  (null? (stack-content s)))
```

To support interfaces, PLT Scheme offers an `interface` form, plus a `class*` variant of `class` that includes a sequence of expressions for interfaces. An interface consists of a collection of method names to be implemented by a class, and like a class, it is a first-class object. As in Java, an interface can extend multiple interfaces.

[1]The class form also supports private field declarations, but we omit them for brevity.

```
expr  =  ...
      |  (class* super-expr
           (interface-expr ...)
           clause ...)
      |  (interface (super-expr ...) name ...)
```

The `generic` form accepts an interface in place of a class, but the current implementation of generics offers no performance advantage for interfaces.

3.2 PLT Scheme vs. Java

If PLT Scheme's class system were not already Java-like, we would have implemented new forms via macros to support Java-to-Scheme compilation. This layering allows us to develop and test the core class system using our existing infrastructure for Scheme, including debugging and test support.

PLT Scheme's class system does not include inner classes or static methods, so the Java-to-Scheme step transforms those Java constructs specially. Static methods are easily converted to procedures, and inner classes have strange scoping rules that seem better handled before shifting to macro-based expansion. Similarly, Java's many namespaces are transferred into Scheme's single namespace by the compiler, rather than by macros. In other words, we use macros to implement the parts of the compiler that fit naturally with lexical scope and local expansion, and we perform other tasks in the compiler.

4 Compilation Model

A single Java source file typically contains one `public` class (or interface). Often, the file itself corresponds to a compilation unit, so that one `.java` file can be compiled to one `.class` (or, in our case, to one `.scm` file).

In general however, reference cycles can occur among `.java` files, as long as they do not lead to inheritance cycles. Thus, the compilation unit corresponds to several mutually dependent `.java` files. For example, one class may refer to a field of another class, and compiling this reference requires information about the structure of the referenced class. In contrast, merely using a class as an identifier's type does not necessarily require information about the class, especially if the identifier is not used.

More concretely, the code in Figure 1 corresponds to three source files, one for each class. Compiling `Empty` requires knowledge of the superclass `List`, while compiling `List` requires knowledge of `Empty` for the constructor call. Similarly, `List` refers to `Cons` and `Cons` refers to `List`. Thus the three classes must all be compiled at the same time. This kind of cyclic reference appears frequently in real Java code.

Java's packages are orthogonal to compilation units because a group of mutually dependent `.java` files might span several Java `package`s. Furthermore, a mutually dependent group of files rarely includes all files for a package, so forcing a compilation unit to be larger than a package would lead to needlessly large compilation units. Finally, in most settings, a Java package can be extended by arbitrary files that simply declare membership in the package, which would cause an entire package to recompile unnecessarily.

To a first approximation, our Java-to-Scheme compiler produces a single Scheme `module` for each collection of mutually dependent Java sources, where `module` is the unit of compilation for PLT Scheme code [7]. Each class used by, but not a member of, the dependent group is `require`-ed into the `module`. The Java specification [8] requires that each class be initialized and available prior to its first use, which the `require` statement ensures.

The `module` is also a unit of organization at the Scheme level, and for interoperability, we would like to maintain the organization of the Java library in the Scheme program. Thus, our Java-to-Scheme compiler actually produces $N+1$ modules for N mutually dependent Java sources: one that combines the Java code into a compilation unit, and then one for each source file to re-export the parts of the compilation unit that are specific to the source.[2] Thus Scheme and Java programmer alike import each class individually. For example, compiling Figure 1 results in four modules: A composite module that contains the code of all three classes and exports all definitions, a `List` module that re-exports `List` and `main`, an `Empty` module that re-exports `Empty`, and a `Cons` module that re-exports `Cons` and field-relevant information.

In practice, we find that groups of mutually dependent files are small, so that the resulting compilation units are manageable. This is no coincidence, since any Java compiler would have to deal with the group as a whole. In other words, this notion of compilation unit is not really specific to our compiler. Rather, having an explicit notion of a compilation unit in our target language has forced us to understand precisely what compilation units are in Java, and to reflect those units in our compiler's result.

Currently, our compiler produces an additional file when generating Scheme from Java code. The extra file contains Java signature information, such as the types and names of fields and methods in a class, which the compiler needs to process additional compilation units. Other Java compilers typically store and access this information in a `.class` directly, and in a future version of our compiler, we intend to explore storing this compile-time information in a `module` in much the same way that compile-time macros are stored in `modules`.

5 Compilation Details

Our compiler begins by parsing Java code using a LEX-/YACC-style parser generator. Source tokens are quickly converted into location-preserving syntax identifiers, as used in macros. Thus, as the generated Scheme code is processed by the Scheme compiler, source information from the original Java program can be preserved during Scheme compilation. This source-location information is used mainly by DrScheme tools or for reporting run-time errors.

As our primary motivation for this work (pedagogic Java subsets) requires control over all error messages reported from the compiler, we chose to compile Java source instead of Java bytecode. While this limits the libraries available to our system, in the future we can use existing bytecode interpreting libraries to alleviate this limitation.

Java and PLT Scheme both strictly enforce an evaluation order on their programs. Coincidentally, both enforce the same ordering on function arguments and nested expressions. Therefore, those Java

[2] If a class is not a member of any dependency cycle, then the compiler produces only one module.

```
abstract class List {
  abstract int length();

  static void main() {
    Test.test(new Empty().length(), 0);
    Test.test(new Cons(1,
                       new Empty()).length(),
              1);
  }
}

class Empty extends List {
  int length() { return 0; }
}

class Cons extends List {
  int car;
  List cdr;
  Cons( int c, List cdr ) {
    this.car = c;
    this.cdr = cdr;
  }
  int length() { return 1 + cdr.length(); }
}
```

Figure 1. A Cyclic Java program

constructs which differ from Scheme only in syntax have a straightforward translation. For example,

```
int a = varA + varB, b = varA - varB;
if (a+b <= 2)
 res = a;
else
 res = b;
```

translates into

```
(let ((a (+ varA varB))
      (b (- varA varB)))
  (if (<= (+ a b) 2)
      (set! res a)
      (set! res b)))
```

wrapped with the appropriate source location and other information. Indeed, the majority of Java's statements and expressions translate as expected.

Currently, mathematical operations directly use standard Scheme operations where possible. Thus, unlike the Java specification, numbers do not have a limited range and will automatically become bignums. In the future, our compiler will use mathematical operations that overflow as in the Java specification.

5.1 Classes

A Java class can contain fields, methods, nested classes (and interfaces), and additional code segments, each of which can be static. Our Scheme class is similar, except that it does not support static members. Nevertheless, a static member closely corresponds to a Scheme function, value, or expression within a restricted namespace, i.e., a `module`, so static Java members are compiled to these scheme forms.

An instance of a class is created with the `new` form described in Section 3.1. As noted in that section, PLT Scheme's `new` triggers the evaluation of the expressions in the top level of the class body. These expressions serve the same purpose as a single Java constructor. However, a Java class can contain multiple constructors, preventing a direct translation from a Java constructor to a sequence of top-level expressions. Instead, we translate Java constructors as normal methods in the Scheme class, and we translate a Java `new` expression into a Scheme `new` followed by a call to a constructor method. This behavior adheres to the guidelines for class instantiation provided by Java's specification [8].

5.2 Fields & Methods

Non-`static` Java fields translate into Scheme `field` declarations. A `static` Java field, meanwhile, translates into a Scheme top-level definition. Thus, the fields

```
static int avgLength;
int car;
```

within the class `Cons` become, roughly

```
(define avgLength 0)
```

and

```
(define Cons
  (class ···
    (field (car 0)) ···))
```

However, the above translation does not connect the variable `avgLength` to the containing class `Cons`. If multiple classes within a compilation unit contain a static field `avgLength`, the definitions would conflict. For non-static fields, Scheme classes do not allow subclasses to shadow field names again potentially allowing conflicts. Additionally, to avoid conflicts between Java's distinct namespaces for fields, methods, and classes, we append a `~f` to the name. Therefore, we combine `avgLength` with the class name and `~f`, forming the result as `Cons-avgLength~f`, and `car` becomes `Cons-car~f`. Note that Scheme programmers using this name effectively indicate the field's class.

Compilation generates a mutator function for both of these fields, plus an accessor function for the instance (non-`static`) field. Since the `module` form prohibits mutating an imported identifier, the mutator function `Cons-avgLength-set!` provides the only means of modifying the static field's value. If the static field is `final`, this mutator is not exported. Also, instance field mutators are not generated when they are `final`. Thus, even without compile-time checking, Scheme programmers cannot violate Java's `final` semantics.

Similarly, instance methods translate into Scheme methods and static methods into function definitions with the class name appended, but the name must be further mangled to support overloading. For example, the class `List` in Figure 2 contains two methods named `max`, one with zero arguments, the other expecting one integer. The method `max(int)` translates into `max-int`, and `max` translates into `max`. This mangling is consistent with the Java bytecode language, where a method name is a composite of the name and the types of the arguments. Also, since "-" may not appear in a Java name, our convention cannot introduce a collision with any other methods in the source.[3]

[3]We do not add a `-m` to method names, because `~f` distinguishes fields from methods, and method and class names must be distinguished already at the Java source level.

```
abstract class List {
 abstract int max();
 abstract int max(int min);
}
```

Figure 2. Overloaded methods

As mentioned in Section 5.1, constructors are compiled as methods, which we identify with special names. The constructor for `Cons` in Figure 1 translates into `Cons-int-List-constructor`. The `-constructor` suffix is not technically necessary to avoid conflicts, but it clarifies that the method corresponds to a constructor.

A `private` Java member *does not* translate to a `private` Scheme member, because `static` Java members are not part of the Scheme class, but Java allows them to access all of the class's members. We protect `private` members from outside access by making the member name local to a module with `define-local-member-name`; the Java-to-Scheme compiler ensures that all accesses within a compilation unit are legal. Our compiler does not currently preserve protection for `protected` and package members.

5.3 Statements

Most Java statements (and expressions) translate directly into Scheme. The primary exceptions are `return`, `break`, `continue`, and `switch`, which implement statement jumps. For all except `switch`,[4] we implement these jumps with `let/cc`:[5]

```
(define-syntax let/cc
  (syntax-rules ()
    ((let/cc k expr ...)
     (call-with-current-continuation
       (lambda (k) expr ...)))))
```

A `return` translates into an invocation of a continuation that was captured at the beginning of the method. For example, the method `length` from `Empty` in Figure 1 becomes

```
(define/public length
  (lambda ()
    (let/cc return-k
      (return-k 0))))
```

The statements `break` and `continue` terminate and restart a `for`, `while`, or `do` loop, respectively. To implement these, we capture suitable continuations outside and inside the loop, such that

```
while(true) {
  if (x == 0)
    break;
  else if (x == 5)
    continue;
  x++;
}
```

becomes

[4]We have not implemented `switch`.

[5]We actually use `let/ec`, which captures an escape-only continuation.

```
(let/cc break-k
  (let loop ()
   (let/cc continue-k
     (when #t
           (if (= x 0)
               (break-k)
               (if (= x 5)
                   (continue-k)
                   (set! x (+ x 1))))
           (loop)))))
```

As it happens, `let/cc` is expensive in PLT Scheme. We plan to apply a source-to-source optimizer to our Java-to-Scheme compiler's output to eliminate these `let/cc` patterns, putting each statement in a separate `letrec`-bound function and chaining them. Although we could avoid `let/cc` in the output of our Java-to-Scheme compiler, it is easier to translate most Java statements directly to Scheme, and then work with Scheme code to optimize.

5.4 Native Methods

Most Java implementations use C to provide native support. Our system, naturally, uses Scheme as the native language. When our compiler encounters a class using `native` methods, such as

```
class Time {
 static native long getSeconds(long since);
 native long getLifetime();
}
```

the resulting `module` for `Time` `requires` a Scheme module *`Time-native-methods`* which must provide a function for each native method. The name of the native method must be the Scheme version of the name, with *`-native`* appended at the end. Thus a native function for `getSeconds` should be named *`Time-getSeconds-long-native`* and `getLifetime` should be *`getLifetime-native`*.

Within the compiled code, a stub method is generated for each native method in the class, which calls the Scheme native function. When `getSeconds` is called, its argument is passed to *`Time-getSeconds-long-native`* by the stub, along with the class value, relevant accessors and mutators, and generics for private methods. An instance method, such as `getLifetime`, additionally receives `this` as its first argument.

5.5 Constructs in Development

We have not completed support of `switch`, labeled statements, nested classes, and reflection. The first two are straightforward, and we discuss our design of the other two further in this section. Our partial implementation of nested classes suggests that this design is close to final.

5.5.1 Nested Classes

In Java, a nested class may either be `static` or an instance class, also known as an inner class. An inner class can appear within statement blocks or after `new` (i.e. an anonymous inner class).

Static nested classes are equivalent to top-level classes that have the same scope as their containing class, with the restriction that they may not contain inner classes. These can be accessed without directly accessing the containing class. When compiled to Java bytecodes, nested classes are lifted out and result in separate `.class` files. We equivalently lift a nested class, and provide a separate module for external access. We treat a nested class and its container as members of a cycle, placing both in the same module.

Inner classes are also compiled to separate classes. Unlike static nested classes, they may not be accessed except through an instance of their containing class. A separate module is therefore not provided, and construction may only occur through a method within the containing class.

The name of a nested class is the concatenation of the containing class's name with the class's own name. Class `B` in

```
class A {
  class B {
  }
}
```

is accessed as `A.B`. For anonymous inner classes, we intend to follow the bytecode strategy: the class will be given a name at compile-time, the containing class name appended with a call to `gensym`, and then lifted as other nested classes.

5.5.2 Reflection

Java supports multiple forms of reflection: examining and interacting with classes and objects specified at runtime; dynamically extending classes; and modifying the means of class loading and compilation. The first one can be supported either with macros or generating methods during compilation to provide the data. We do not yet know how the second will be supported, or what support for the third would mean within our system.

The first form of reflection allows users to create new class instances with strings, inspect and modify fields, call methods, and inspect what fields and methods are available. The last of these is easily supported by generating the information during compilation and storing it in an appropriate method. The other functionality can be supported through Scheme functions.

6 Run-Time Support

Java provides two kinds of built-in data: primitive values, such as numbers and characters, and instances of predefined classes. The former translate directly into Scheme, and most of the latter (in `java.lang`) can be implemented in Java. For the remainder of the built-in classes, we define classes directly in Scheme.

6.1 Strings

Although the `String` class can be implemented in Java using an array of `char`s, we implement `String` in Scheme. This implementation allows a Scheme string to hold the characters of a Java string, thus facilitating interoperability. From the Scheme perspective, a Java `String` provides a *`get-mzscheme-string`* method to return an immutable Scheme string.

6.2 Arrays

A Java array cannot be a Scheme vector, because a Java array can be cast to and from `Object` and because assignments to the array indices must be checked (to ensure that only objects of a suitable type are placed into the array). For example, an array created to contain

`List` objects might be cast to `Object[]`. Assignments into the array must be checked to ensure that only `List`, `Cons`, and `Empty` objects appear in the array.

To allow casts and implement Java's restrictions, a Java array is an instance of a class that descends from `Object`. The class is entirely written in Scheme, and array content is implemented through a private `vector`. Access and mutation to the `vector` are handled by methods that perform the necessary checks.

6.3 Exceptions

PLT Scheme's exception system behaves much like Java's. A value can be raised as an exception using *`raise`*, which is like Java's `throw`, and an exception can be caught using `with-handlers`. The `with-handlers` form includes a predicate for the exception and a handler, which is analogous to Java's implicit instance test with `catch` and the body of the `catch` form. The body of a `with-handlers` form corresponds to the body of a `try` before `catch`. We implement Java's `finally` clause using `dynamic-wind`.

Unlike Java's `throw`, the PLT's *`raise`* accepts any value, not just instances of a throwable. Nevertheless, PLT tools work best when the raised value is an instance of the `exn` record. This record contains fields specifying the message, source location of the error, and tracing information.

Our implementation of the `Throwable` class connects Java exception objects to PLT Scheme exception records. A `Throwable` instance contains a PLT exception record, and when the `Throwable` is given to `throw`, the exception record is extracted and raised. This exception record is an extension of the base PLT exception record, with an added field referencing the `Throwable` instance. If a `catch` form catches the exception, the `Throwable` can be extracted.

Besides generally fostering interoperability, this re-use of PLT Scheme's exception system ensures that Java programs running within DrScheme get source highlighting and stack traces for errors, etc. All of Java's other built-in exception classes (which derive from Throwable) are compiled from source.

7 Interoperability

Java–Scheme interoperability is not seamless in our current implementation, but programs written in one language can already access libraries written in the other.

7.1 Java from Scheme

A compiled Java library is a `module` containing Scheme definitions, so that importing the library is therefore as simple as importing a Scheme library. Scheme programmers gain access to the class, (non-private) static members, field accessors, and nested classes of the Java code, and they can derive new classes and interfaces from the Java classes and interfaces. In general, they may treat bindings from Java code without regard to the original language, except to the degree that data types and protocols expose that language.

In particular, to interact with Java classes, a Scheme programmer must remember certain protocols regarding constructors and inner classes. As discussed in Section 5.1, the constructor must be called after an object is instantiated, which means that the programmer must explicitly invoke the constructor when instantiating or extending the class. Inner classes must not be instantiated directly with `new`, but instead instantiated through a method supplied by the containing class. (In all probability we can make inner classes module-local, and expose only an interface for instance tests, but we are uncertain whether this strategy will work with reflection.)

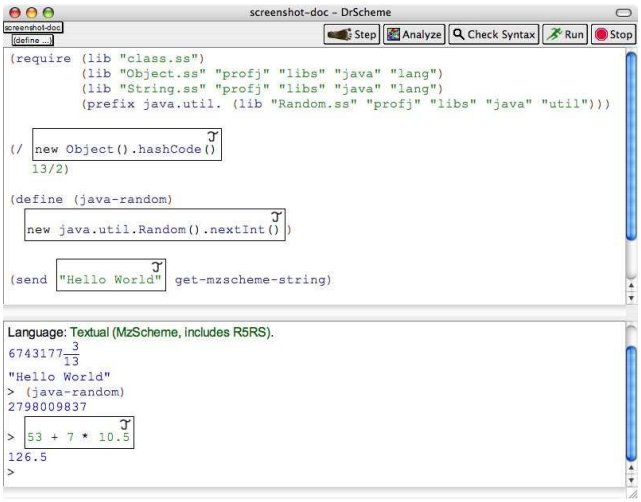


Figure 3. Java Box

As a practical matter, a Scheme programmer will think of a Java-implemented library in Java terms, and therefore must manually mangle member names, as discussed in Section 5.2. Mangled names can potentially be quite long. Consider the method `equals`, which takes an instance of `Object`. The mangled version is *`equals-java.lang.Object`*, to fully qualify which `Object` is meant. We are investigating ways to avoid this problem.

One strategy, which presently partially works within DrScheme, is to insert a graphical box representing a Java expression (see Figure 3), instead of plain text. The expression within the box contains Java instead of Scheme and results in a value. Assigning types to the arguments (to resolve overloading) remains an open problem, thus Scheme values cannot be accessed within a box.

Another remaining problem is that, while our compiler is designed to produce modules that are compatible with PLT Scheme's compilation model, the compilation manager itself does not know how to invoke the compiler (given a reference to a `.java` file). We are working on an extension of the compilation manager that locates a compiler based on a file's suffix. For now, manual compilation meets our immediate needs.

7.2 Scheme from Java

The `native` mechanism described in Section 5.4 provides a way to make Scheme functionality available to Java, but `native` is not a suitable mechanism for making a Scheme class available as a Java class. Instead, our compiler can use a Scheme class directly as a Java class, for instantiation, extension and overriding, or instance tests. At compile time, the compiler needs specific type information for the class, its fields, and its methods. This information is currently supplied in a separate file, with the extension `.jinfo`.

Every class in Java extends `Object`, but not every Scheme class does so. To resolve this mismatch, the compiler does not actually treat `Object` as a class. Instead:

- The core `Object` methods are implemented in a mixin, *`Object-mixin`*. Therefore, `Object` methods can be added to any Scheme class that does not already supply them, such as when a non-`Object` Scheme class is used in Java.
- Indeed the `Object` class used for instantiation or class extension in Java code is actually (*`Object-mixin`* `object%`).
- `Object` instance tests are implemented through an interface, instead of a class. This works because `Object` has no fields (fortunately) so the class is never needed.

A `.jinfo` file indicates whether a Scheme class already extends `Object` or not, so that the compiler can introduce an application of *`Object-mixin`* as necessary. A Scheme class can explicitly extend a use of *`Object-mixin`* to override `Object` methods.

We used the native interface to quickly develop a pedagogic graphics library, based on existing Scheme functionality. Java programmers are presented with a canvas class, which supports drawing various geometric shapes in a window. This class can be subclassed with changes to its functionality. Internally, the Java class connects to a functional graphics interface over MrEd's graphics.

8 Performance

So far, we have invested little effort in optimizing the code that our compiler generates. As a result, Java programs executed through our compiler perform poorly compared to execution on a standard JVM. In fact, Java programs perform poorly even compared to equivalent programs written directly in Scheme. The current performance problems have many sources (including the use of continuations, as noted in Section 5.3), all of which we expect to eliminate in the near future. Ultimately, we expect performance from Java code that is comparable to that of PLT Scheme code.

9 Related work

The J2S compiler [3] compiles Java bytecodes into Scheme to achieve good performance of Java-only programs. This compiler additionally targets Intel X86 with its JBCC addition. J2S globally analyzes and optimizes the bytecode to enhance performance. Java classes compile into vectors containing method tables, where methods are implemented as top-level definitions. Instances of a class are also represented as vectors. Unlike our system, this compilation model does not facilitate conceptual interoperability between Scheme and Java programs. Native methods may be written in Scheme, C, C++, or assembly, which allows greater flexibility than with our system at the cost of potential loss of security. As with our system, J2S does not support reflection.

Several Scheme implementations compile to Java (either source or bytecode) [1, 2, 4, 12, 15]. All of these implementations address the interaction between Scheme and Java, but whereas we must address the problem of handling object-oriented features in Scheme, implementors of Scheme-to-Java implementors must devise means of handling closures, continuations, and other Scheme data within Java:

- JScheme [1, 2] compiles an almost-R^4RS Scheme to Java. Within Scheme, the programmer may use static methods and fields, create instances of classes and access its methods and fields, and implement existing interfaces. Scheme names containing certain characters are interpreted automatically as manglings of Java names. Java's reflection functionality is employed to select (based on the runtime type of the arguments) which method to call. This technique is slower than selecting the method statically, but requires less mangling.
- SISC [11] interprets R^5RS, with a Java class representing each kind of Scheme value. Closures are represented as Java instances containing an explicit environment. Various SISC methods provide interaction with Java [12]. As with JScheme the user may instantiate Java objects, access methods and fields, and implement an interface. When passing Scheme values into Java programs, they must be converted from Scheme objects into the values expected by Java, and vice-versa. To access Scheme from Java, the interpreter is invoked with appropriate pointers to the Scheme code.
- The Kawa [4] compiler takes R^5RS code to Java bytecode. Functions are represented as classes, and Scheme values are represented by Java implementations. Java static methods may be accessed through a special primitive function class. Values must be converted from Kawa specific representations into values expected by Java. In general, reflection is used to select the method called, but in some cases, the compiler can determine which overloaded method should be called and specifies it statically.
- In addition to a C back end, Bigloo [14, 15] also offers a bytecode back end. For this, functions are compiled into either loops, methods or classes (to support closures). Scheme programmers may access and extend Java classes.

PLT Scheme developers have worked on embedding other languages in Scheme, including Python [10], OCaml, and Standard ML. At present, the Java-to-Scheme compiler described here is the most complete.

10 Conclusion

Our strategy for compiling Java to Scheme is straightforward: we first develop macro-based extensions of Scheme that mirror Java's constructs, and then we translate Java code to the extended variant of Scheme. This strategy facilitates interoperability between the two languages. It also simplifies debugging of the compiler, since the compiler's output is human-readable, and the target macros can be developed and tested independently from the compiler.

For PLT Scheme, the main target constructs for the compiler are `module` and `class` (plus standard Scheme constructs, such as procedures). These forms preserve most of the safety and security properties of Java code, ensuring that the Java programmer's expected invariants hold when the code is used by a Scheme programmer. Scheme programmers must follow a few protocols when interacting with Java libraries, and manually include type information within method calls. However, we believe that future work will reduce these obstacles.

While still in development, our Java-to-Scheme compiler has deployed with PLT Scheme since version 205. We continue to add language constructs and interoperability features.

Acknowledgments

We would like to thank Mario Latendrese, whose Java to Java bytecode compiler, written in Scheme, provided the front-end for preliminary versions of this work. We would also like to thank Matthias Felleisein for comments on early drafts of this paper, and the reviewers for their many helpful comments.

11 References

[1] K. Anderson, T. Hickey, and P. Norvig. *JScheme User manual*, Apr. 2002. `jscheme.sourceforge.net/jscheme/doc/userman.html`.

[2] K. R. Anderson, T. J. Hickey, and P. Norvig. SILK - a playful blend of Scheme and Java. In *Proc. Workshop on Scheme and Functional Programming*, Sept. 2000.

[3] É. Bergeron. Compilation statique de Java. Master's thesis, Université de Montréal, 2002.

[4] P. Bothner. Kawa: Compiling Scheme to Java. In *Lisp Users Conference*, Nov. 1998.

[5] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, Sept. 1997.

[6] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ACM International Conference on Functional Programming*, pages 94–104, Sept. 1998.

[7] M. Flatt. Composable and compilable macros: You want it when? In *Proc. ACM International Conference on Functional Programming*, pages 72–83, Oct. 2002.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[9] K. E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177, Oct. 2003.

[10] P. Meunier and D. Silva. From Python to PLT Scheme. In *Proc. Workshop on Scheme and Functional Programming*, Nov. 2003.

[11] S. G. Miller. SISC: A complete Scheme interpreter in Java. `sisc.sourceforge.net/sisc.pdf`, Feb. 2003.

[12] S. G. Miller and M. Radestock. *SISC for Seasoned Schemers*, 2003. `sisc.sourceforge.net/manual`.

[13] T. Parr. ANTLR parser generator and translator generator. `http://www.antlr.org/`.

[14] B. P. Serpette and M. Serrano. Compiling Scheme to JVM bytecode: A performance study. In *Proc. ACM International Conference on Functional Programming*, Oct. 2002.

[15] M. Serrano. *Bigloo: A "practical Scheme compiler" User Manual*, Apr. 2004. `www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.

[16] K. Siegrist. Virtual laboratories in probability and statistics. `http://www.math.uah.edu/stat/`.

Foreign Interface for PLT Scheme

Eli Barzilay
Northeastern University

Dmitry Orlovsky
Northeastern University

Abstract

Even a programmer devoted to Scheme may prefer using foreign libraries in certain situation. Connecting the two worlds involves glue code, usually using C, which requires significant programming efforts and system expertise. In this paper we describe a PLT Scheme extension for interacting with foreign code, designed around a simple philosophy: *stay in the fun world*, even if it is no longer a safe sand box. Our system relieves the programmer from low-level technicalities while keeping the benefits of Scheme as a better programming environment compared to C.

1 Introduction

Scheme has proved itself as a useful and fun language, good for both general-purpose and domain-specific usages. However, schemers cannot assume a closed system; other languages will always exist, leading to a need for interfacing with functionality that is accessible through foreign libraries. Such libraries come in many different flavors, but the popular 'least common denominator' has been, and still is, plain C libraries[1]. Our goal is to create a mechanism within Scheme for smooth interfacing with such foreign libraries.

1.1 Background

A foreign interface is a piece of *glue code*, intended to make it possible to use functionality written in one language (often C) available to programs written in another (usually high-level) language. Such glue code involves low-level details that users of high-level languages usually take for granted. For example:

- marshaling objects to and from foreign code,
- managing memory and other resources,
- dealing with different calling conventions, implicit function arguments, etc.

[1] Different languages can be used to create foreign libraries, "C" is only used as a generic label.

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

Foreign function interfaces are subsystems that create such glue code, simplifying an otherwise tedious and error-prone task.

1.2 Foreign Interfaces

There are lots of existing foreign function interfaces; Urban's FFI survey [17], although an incomplete project, provides a good discussion of such systems and relevant issues. Generally speaking, such interfaces can be classified as either static code generators or dynamic foreign interfaces. In principle, the two are quite similar:

- A *static* foreign interface is created and compiled statically, *before* running the program that intends to use it;
- A *dynamic* interface is created at run-time, *while* the application is running.

In practice, the differences are more dramatic:

- A static interface is usually implemented using a C compiler. The advantage of this approach is that it is easy interface foreign code, as most of it is intended to be linked in using a compiler (for example, C header files are used to describe an interface), and since most languages are implemented in C, they provide convenient facilities for calling C functions. Disadvantages of the static approach include being restricted to the pre-compiled interface, requiring either a compiler or a platform dependent binary distribution for such code.
- A dynamic interface is generated at run-time, leading to the obvious advantage of requiring no C compiler or binary distributions. This has a significant effect on dynamic languages like Scheme, where single running REPL can be used to connect to different libraries, supporting exploratory programming in a natural way. The disadvantage of this approach is that it requires more (platform-dependent) low-level work such as stack management and creating stubs (glue functions), while not getting the usual support from a C compiler.

The issues that need addressing are essentially the same ones described in Section 1.1, only the approach differs. The technical issues involved in an interface implementation make static interface generators more popular. It should be noted that it is common to call these systems "foreign *function* interfaces" — in the following text we prefer "foreign interfaces" as these interfaces deal with accessing foreign objects as well as foreign calls.

In both the static and the dynamic cases, it is desirable to have some description of the foreign entities, usually functions, in a way that can help automate the process of generating the glue layer. In this context a "function" can be viewed differently depending on your point of view: from the low-level side, a function is simply a pointer and a description of how it is called; from the high-level Scheme

side, it is an object that is expected to have the usual function semantics. *Interface description languages* (IDLs) have a major role in foreign interface systems — these are languages that express arbitrary function behaviors for both of these viewpoints:

- On the C side, there is the type definition of the function, and possibly additional information such as input/output pointers, object ownership, etc.
- In addition to this, there are details that are related to the Scheme side. For example, automatic memory management issues, value marshaling, dealing with aggregates (vectors and structs), and creating new object types.
- On the Scheme side, the result is a plain procedure, like any other Scheme procedure object.

Ideally, the IDL that is used to describe the interface is rich enough to express both views while providing enough information to completely automate the interface generation.

1.3 Implementing a Dynamic Interface

The low-level mechanics of foreign function calls are usually very demanding: managing functions at the binary level is inherently platform dependent, and can even require assembly code or other compiler-specific hacks. Statically, these problems are not too difficult: simply generate C glue code, and let the C compiler do its usual work. Doing this efficiently in a dynamic fashion is difficult, since it is usually not desirable to drag a complete C compiler into your run-time. Dealing with the dynamic aspects of foreign functions is greatly simplified using a library that handles the low-level details: we use `libffi` [11], a library that supports foreign function call-outs and call-backs.

- A call-out is a normal function call. In a dynamic setting, we create a "call-interface" object which specifies (binary) input/output types; this object can be used with an arbitrary function pointer and an array of input values to perform a call-out to the function and retrieve its result. Doing this requires manipulating the stack and knowing how a function is called, these are details that `libffi` deals with.
- A call-back is trickier. Our Scheme implementation has several fixed C-level functions which can implement arbitrary Scheme evaluation. A callback is, however, a simple function pointer — no additional information is available. Modern systems (e.g, Gnome) that use callbacks allow user to register a function pointer together with an arbitrary data pointer, but there is no standard way for this. A proper solution is one that allows creating general "C closures" — combining a function and a data pointer into a single new function pointer. Again, this is technically challenging, as it requires generating stub functions at run-time, which, when applied to some arguments, call the packaged function with the packaged data pointer and the arguments. Again, `libffi` provides the required magic.

`libffi` is maintained and distributed as part of the GCC project, but its goal is to provide a portable library. We use it for all platforms that PLT Scheme targets, including Windows (using a slightly adapted version that works with Microsoft's compiler, courtesy of the Thomas Heller [13]).

1.4 Outline

In Section 2 we state the goal of our work, emphasizing our main design principle. Section 3 describes our implementation, both the C part of the code and the complementing Scheme module. Section 4 demonstrates how our system copes with some of the common and uncommon situations that interface programmers deal with. We conclude with a related work comparison, and outline future plans.

2 Goal: Use Foreign Libraries, Avoid C

Our design follows a simple principle: keep C-level functionality to a minimum. The core of a system for interfacing foreign libraries must itself be written in C, but we try to make such functionality available to Scheme as soon as possible, putting more responsibility on the Scheme level. When dealing with the many details of the interface, mainly type declarations and data marshaling, there is a natural tendency to make a system that is rich in features. We avoid dealing with such complexities in C when possible, providing just enough of an interface that makes it possible to do it in Scheme instead. The combination of a dynamic interface and a minimalistic C-level implementation that should be complemented by Scheme code are the main features that make our approach unique.

Switching more responsibility to Scheme comes with benefits that are familiar to Scheme programmers, but there is an additional advantage that is important in this particular case: the important issue is generating glue code that bridges the gap between foreign libraries and the high-level language. In the static case this involves either complex yet limited C preprocessor acrobatics (e.g., SWIG [1] goes as far as implementing its own C parser). On the other hand, Scheme already comes with a superior syntax system, and PLT Scheme makes this even better with additional language features (syntax objects, module system, etc). This syntax system is much easier for implementing sophisticated glue code with, especially considering our target crowd which undoubtedly feels more at home in the Scheme world.

For example, consider the issue of primitive foreign types that are handled by an interface. Once we can move C integers from Scheme to C and back, we might consider extending the system to deal with C enumerations. This raises a few questions regarding the interface design — how should this C definition:

```
typedef enum { foo1, foo2, foo3 } foo;
```

be available for Scheme code?

- Should we provide three integer bindings? If so, how do we deal with name clashes?
- Otherwise, should we use a mapping from strings to integers? Maybe use symbols? What about enumerated values that are or-able bit patterns? How should such a map be implemented: as a linked list? A vector of constant names? A hash table?

Answers to these questions determine the nature of the C implementation; once it is written, trying alternatives lead to significant maintenance costs. Our design keeps such complications away from the C level, pushing them up to the Scheme side where there are better ways to deal with them. For example, the C level part of our interface does not commit to a specific implementation for enumerations — it simply exposes C integers. Different strategies are then implemented in the Scheme part, resulting in easier code maintenance. In addition, some Scheme aspects are less accessible from C, making a Scheme solution even more attractive. for example, implementing enumerations as bindings that use the module system to avoid global name-space pollution, or implementing them as syntax objects (removing run-time lookup costs) are both much harder to implement in C than in Scheme.

Another important factor in the complexity of the C implementa-

tion is the issue of safety. Scheme is a *safe* language — as buggy as your code might be, you never expect the Scheme process to crash: if such a crash happens, the blame is in the language implementation. Using C extensions such as the ones that PLT Scheme always had, changes things a little — the code to blame can be either in the language implementation, or in the C extension. The invariant fact is that Scheme code can never be blamed for such crashes — they are exclusively considered a C-level problem. There is therefore a yellow caution tape around code that can be blamed for such crashes: it lies exactly on the language boundaries, C on one side and Scheme on the other.

A dynamic foreign function interface inevitably breaks this property: bad Scheme code that defines an interface to a foreign function can specify an integer argument where a pointer is expected, leading to a crash (at best). Using dynamic interface systems does not seem so bad though — a foreign function definition is written in Scheme, but conceptually it is perceived as part of the C world. Scheme code, with the exception of such definitions, is still as safe as it has always been, the yellow caution tape is moved just a little so it surrounds Scheme definitions of foreign interfaces too. This point drives a dynamic foreign interface system to try to be as safe as possible: if function interfaces are the only things that can lead to crashes, then it is desirable to make the system safe in all other respects. For example, when dealing with pointers (arrays referencing, allocations, garbage collection) safety issues go in the C code, making it much more complicated than it would otherwise be.

In contrast, our implementation extends traditional dynamic interface systems by exposing more 'dangerous' operations. Functionality that had to be part of the C world is now accessible in Scheme, moving the yellow tape again to encompass more Scheme code. The average programmer is not concerned with this extra functionality, but interface implementors can now deal with more foreign code without leaving Scheme. Many design decisions that usually affect the C interface can now be pushed up to the Scheme level.

The issue of safety is now related to the module system: the new foreign interface bindings are enclosed in a module. If a Scheme process crashes, the blame is either on C code, or on Scheme code that uses this module: such code is therefore taken as substituting C code, potentially suffering from C's usual illnesses. Code that does not use this module is expected to be as safe as it previously was.

To summarize, the yellow caution tape surrounds more Scheme code now: it lies at the C/Scheme language border *except* for code that uses the new module which is inside the tape. In essence, using the a Scheme module is similar to a using Modula-3's [12] `UNSAFE` keyword to declare unsafe code. Quoting Harbison [12, Section 13.3.1] from the Modula-3 book:

> Modula-3 also provides unsafe features, but it differs from many other languages in isolating those features. The unsafe language features are accessible only in interfaces and modules that are labeled by the keyword `UNSAFE`. [...] When all modules and interfaces are safe, Modula-3 guarantees that there will be no unchecked run-time errors. By introducing `UNSAFE`, the programmer assumes part of that burden.

Our system is slightly different in that Scheme modules can provide additional functionality for interface writers, meaning that they will not provide a safe interface, making them have a status similar to that of the new module. This means that rather than a fixed set of unsafe language features, we have a system where these features can also be extended.

An example of this design philosophy is our use of pointers. First, a new Scheme pointer object is introduced, then low-level functions that deal with pointers are added. These are procedures that allocate memory blocks (using one of several '`malloc`' variants), free blocks (for GC-invisible blocks), reference pointers, and set values at a pointer locations. This new functionality is useful in itself, even when there are no foreign libraries to interface with. For example, the procedural part of SRFI 4 [6] can now be trivially implemented in Scheme. Several foreign interfaces have a similar generic 'pointer' object, but it is usually viewed as a last-resort object when an unknown pointer is returned[2] or when an interface is too lightweight for proper types[3] — this is in contrast to our view, where a pointer object is taken as part of the fundamental framework that makes Scheme a viable C substitute for glue code.

3 Implementation

Our implementation consists of a C part, implementing the low-level functionality, and a Scheme part that builds on top of it. The C part of our interface is available as a built-in '`#%foreign`' module which is part of the MzScheme core of PLT Scheme (it is part of the MzScheme executable). This implements the thin interface, providing just enough to make it possible to fill in the gaps using Scheme. This module is therefore intended to be used only by the Scheme part of our interface: the '`(lib "foreign.ss")`' module which is part of MzLib, serving as a wrapper around the internal bindings. For brevity, we refer to the Scheme module as '`foreign`'.

The '`#%foreign`' functionality that is implemented in C is described in Section 3.1, and the Scheme '`foreign`' module is described in Section 3.2.

3.1 The '`#%foreign`' Module: C-Level Interface

The C implementation can be roughly divided into three parts, described in the following sections. Most of this is unrelated to foreign libraries, but providing the framework that make such interactions possible, and making Scheme rich enough to substitute C.

3.1.1 C Types

C-types[4] lie at the core of our system, as they provide the basic specifications for data that is passed on to and back from foreign libraries. We need some way to specify the correlation between tagged Scheme values and the various C types. This mapping is not one-to-one: a single C type can be interpreted as several Scheme types, and a single Scheme type can be translated to different C types. We implement C type objects for this, available as new first-class Scheme values, accessible through '`#%foreign`' bindings. Each C type object has three main parts:

- The actual C type that it represents (a `libffi` type descriptor),
- Code that translates corresponding Scheme objects to C,
- Code that translates such C values to Scheme objects.

In addition, there is some utility information such as a predicate, byte size and alignment. The translation code for these primitive C types is implemented in C. Table 1 presents a summary of the current built-in primitive types[5].

[2] For example, the SWIG manual uses `malloc`, `realloc` and `free` as a simple interface example which uses pointer objects.

[3] Our `cpointer` type pre-existed for PLT Scheme extensions, and was intended for "extensions with modest needs".

[4] Again, "C" is only used as it reflects binary level objects.

[5] The name convention that we have used is that a type called '`foo`' is available in Scheme as a '`_foo`' binding.

Primitive Type	Usage
_void	returns a Scheme void value when used as an output type
_int8, ..., _int64	integer types in various sizes
_uint8, ..., _uint64	non-negative integers
_byte, _word, _int, _uint	aliases for _uint8, _uint16, _int32, and _uint32 respectively
_long, _ulong	aliases for 32- or 64-bit integers, depending on the meaning of 'long' for the current platform
_fixint, _ufixint, _fixnum, _ufixnum	versions of integers (int and long resp.) that assume fitting into an immediate Scheme fixnum integer
_float, _double	floating point numbers (inexacts)
_bool	booleans (as C integers)
_bytes	byte-strings (plain char strings and memory blocks represented as byte-strings)
_string/ucs-4, _string/utf-16	Unicode string types
_path, _symbol	path strings and symbol names as strings (interned when used as an output type)
_pointer	a 'cpointer' object encapsulating a pointer value and an optional tag, #f is used for a NULL pointer
_scheme	a Scheme_Object* pointer, for any Scheme boxed value, this will be its actual pointer

Table 1. Primitive types

Users can create new types in two flavors:

- User-defined types are made by the 'makectype' primitive, and are analogous to primitive types. To create such a type a programmer has to:
  1. Choose the set S_1 of Scheme objects that the new type should handle. This can be any set — combination of several Scheme types, subsets, or a few random values.
  2. Choose an existing C type T as a base type. This type handles some set S_2 of Scheme objects.
  3. Write two procedures: one that translates an S_1 value to S_2 and one that goes the other way.
  4. Apply 'makectype' on T and the two translators.

  When the new C type is used to send values to foreign code (function arguments, or setting pointers), the first translator is used and processing continues with T, and when receiving values from foreign code (return values or pointer references), T is used first and the second translator is then applied. The implementation of user types does not involve libffi, which only sees primitive types.
- New struct types are created from a list of existing types using the 'makecstructtype' primitive. This is mainly implemented by libffi since it describes a new low-level data type with new size and alignment information. On the Scheme side the resulting primitive type is similar to a _pointer, but when it is used to send or receive values, the contents of the pointer is copied rather than the pointer itself.

No additional functionality is implemented at the C level for these types except trivial accessors and size/alignment information. Additional abstraction layers like enumerations and struct constructors and deconstructors are implemented in Scheme. As a result, we don't have to commit to a specific marshaling scheme at the binary level (in fact, the Scheme part of the interface implements two different marshaling schemes for each of these cases).

3.1.2 Pointers

As mentioned above, pointers are an integral part of our interface, exposed as useful Scheme objects. A Scheme pointer object encapsulates the actual pointer value (adding an extra level of indirection), and a 'tag' which is an arbitrary Scheme object. C functionality is limited to a usable minimum: allocating memory blocks (using various allocator functions — either through the garbage collector, or raw malloc), referencing and setting pointed values (given a type), and pointer equality.

Again, functionality implemented by the C level is kept to a minimum. For example, the tag values that are attached to pointers can be used to enforce a type for referencing and setting a pointed value, but such a design can be better implemented and enforced in Scheme, so these tags are *ignored* by the C part of the interface.

3.1.3 Interfacing Foreign Functionality

So far, all C-level functionality is useful by itself, extending Scheme so it can handle machine-level raw data. The final piece of the C part of our interface is the one that actually deals with foreign libraries. First, there is functionality for opening a dynamic library and pulling out objects. These objects can be used as pointer objects, so it is possible to both reference and change their values (useful for libraries that contain user-modifiable customization hooks).

Dealing with function values is separated into function calls that we can do ("callouts") and calls from foreign code to our functions ("callbacks"). This is where libffi makes the implementation much easier. Two Scheme-accessible procedures, fficall and fficallback, are in charge of converting C functions to Scheme (callouts) and Scheme procedures to C (callbacks) respectively. At the Scheme level, these procedures are used by a new _cprocedure type constructor, which provides a symmetric 'marshaling' interface for both ways of this conversion, so users are not aware of any differences in the underlying translation mechanism.

Bindings that are implemented by the C part of our implementation and made available through the '#%foreign' module are listed in Table 2. This, together with Table 1, is a complete summary of the C-level implementation. Again, '#%foreign' is not intended for use outside of our 'foreign' implementation (described next), but many of these procedures are re-exported by 'foreign'.

3.1.4 Garbage Collection Issues

There are some important memory management issues that should be mentioned at this point: a moving garbage collection, such as the one used by the precise PLT Scheme version (mzscheme3m) complicates things considerably when foreign code interacts with objects on the (GC-visible) Scheme heap. There are certain objects that should not move in memory, most notably, the callable function pointers generated by libffi to implement C closures must not move, so we need to take extra care in allocating these using plain malloc, where the garbage collector does not touch them. Callbacks are especially fragile in this aspect: when C code calls Scheme code the garbage collector might be triggered and any GC-visible pointers that the C function might use will inevitably be invalidated. This problem does not have an easy solution — either memory is managed by a non-moving collector possibly managing different memory regions using different collectors (this solution is impossible with PLT Scheme's precise GC), or doing manual management. The C implementation takes care of this when deal-

Primitive Bindings	Usage
`ffi-lib`, `ffi-lib?`, `ffi-lib-name`	open a foreign library and related functionality
`ffi-obj`, `ffi-obj?`, `ffi-obj-lib`, `ffi-obj-name`	get a foreign object pointer from a library and related functionality
`make-ctype`, `make-cstruct-type`, `ctype?`, `ctype-basetype`, `ctype-scheme->c`, `ctype-c->scheme`, `ctype-sizeof`, `ctype-alignof`	Handling C type descriptor objects (see Section 3.1.1)
`cpointer?`, `cpointer-tag`, `set-cpointer-tag!`, `ptr-ref`, `ptr-set!`, `ptr-equal?`	Handling C pointer objects (see Section 3.2.2)
`malloc`, `end-stubborn-change`, `free`, `make-sized-byte-string`, `register-finalizer`	Interface for the standard C `malloc` and other allocators that are used in MzScheme, and related memory management functions
`ffi-call`, `ffi-callback`, `ffi-callback?`	creating a call-out object (a Scheme procedure that calls a foreign function when applied) from a C pointer and creating callbacks (objects that can be passed onto foreign functions as function pointers) from Scheme procedures, both functions accept an input type list and an output type

Table 2. Primitive '`#%foreign`' bindings

ing with `libffi` objects, but nothing else. If a movable pointer is passed on to a C function which can use Scheme callbacks or otherwise retain it, then it is the responsibility of the Scheme level to deal with copying these values to non-movable memory (using the system's raw `malloc` which is accessible in Scheme). The Scheme part of our interface simplifies some of these issues, but there is no general solution when (potentially misbehaved) foreign code is involved, since such code is ignorant of any memory management issues for objects it does not "own".

A related issue is dealing with pointers that can be contained in other objects. The Scheme-visible '`malloc`' function uses atomic allocation by default except for allocating a `_pointer`- or a `_scheme`-based type. User-created `struct` types are, however, problematic because they can hold both pointers and other values. Our implementation uses only atomic memory blocks for these, which works as long as there are no GC-able pointers in `struct`s, which so far was not a problem. We have a plan for dealing with such pointers, in case a solution is needed: expand new struct types with a map of contained GC-able pointer offsets. In any case, users should be aware of the fact that memory blocks are moved and use raw-`malloc`ed pointers as necessary when callbacks or library references are involved.

3.2 The `foreign` Module: Scheme-Level Interface

At the Scheme level, we have added a new '`(lib "foreign.ss")`' module to MzLib. Scheme programmers should use this module which complements the built-in '`#%foreign`' module. The purpose of this module is to re-export some useful parts of '`#%foreign`' with an additional degree of sanity and convenience. For example, '`getffiobj`' is a convenient procedure that combines '`ffilib`' to open a library, '`ffiobj`' to retrieve a pointer, and '`ptrref`' to convert it into a Scheme value. In addition, it builds a layer of additional functionality using the built-in module, varying from new types, through an IDL, to memory management issues.

Defined Type	Usage
`_string/utf-8`, `_string/locale`, `_string/latin-1`	various C strings, using different encoding
`_string`	uses one of the existing string types, depending on the value of the `default_stringtype` parameter; '`#f`' is used as a `NULL` value
`_file`	similar to the `_path` type, except that path names are resolved using `expandpath`
`_string/eof`	similar to `_string`, but in case of `#f` (`NULL`), an end-of-file object is returned
`_enum`, `_bitmask`	these are actually functions that consume a list of symbols, and create an integer-mapping type that translates a single symbol (`_enum`) or a list of symbols (`_bitmask`) to an integer

Table 3. Simple types defined by the Scheme module

3.2.1 Additional Types

The Scheme module, like the C part, revolves mainly around types. First, there are several simple types that are implemented in the Scheme module, summarized in Table 3. Adding these types is simple, as described in Section 3.1.1, for example, the `_file` type is intended to make it easy to interact with foreign functions that expect a file name — making it possible to use names like "`~foo/bar`". The definition in '`foreign`' involves using `expandpath` when going from Scheme to C, and leaving the path as is when going from C to Scheme:

```
(define _file
  (make-ctype  ; create a new type,
   _path       ; based on _path
   expand-path ; expand-path when sent out
   #f))        ; receive: same as _path
```

Since this part of the implementation is in Scheme, we can now develop better solutions than we could if we used only C. For example, note that `_enum` and `_bitmask` are not type objects, but *functions* that create type objects — they are *type constructors*. Also, note that there are multiple string types, since our system is integrated into the development version of PLT Scheme which uses Unicode for its strings — the `_string` type is therefore an 'identifier syntax' that expands into a usage of the '`default_stringtype`' parameter. Both of these would take a much heavier implementation if they were implemented in C.

3.2.2 Pointer Types

Section 3.1.2 mentions that Scheme pointer objects have an arbitrary 'tag' value associated with them, and that these tags are ignored by the C part of the interface. The '`foreign`' module provides a `_cpointer` function that, when given some Scheme value, constructs a new `_pointer`-based type which tags pointer objects when they arrive from the foreign side, and raises an error when passing a pointer with the wrong (non-`eq?`) tag from Scheme. This functionality might be extended in the future to use the tag value in some more meaningful way, for example, make it be another type object and make pointer dereferencing use it instead of taking a type argument, or use it to imitate inheritance where a pointer can be used in places where an ancestor pointer kind is expected. In addition to the `_cpointer` function, there is a `definecpointertype` syntax:

(_define-cpointer-type ⟨*id*⟩
[⟨*type-or-#f*⟩ ⟨*scm→c*⟩ ⟨*c→scm*⟩])

which defines such a type using "⟨*id*⟩" as a tag, together with a '⟨*id*⟩?' predicate and a '⟨*id*⟩-tag' binding for the tag value.

The optional type and translation arguments can be used to specify the base type in case it is not _pointer (for example, if it is a struct type), and translation procedures. Such arguments are also available for _cpointer.

3.2.3 Vector Types

Exposing C functionality in Scheme makes it possible to use arbitrary blocks of memory to hold data. Allocating such a block is even simpler with the provided list->cblock and cblock->list, both implemented in Scheme, but the result is just a bare pointer object. It is therefore useful to encapsulate such a memory block with the type of objects it uses and the number of objects contained in it. Using this we benefit from no per-item storage overhead as well as making some foreign interfaces easier to deal with, and at the same time ensure that there are no violation of the vector bounds. Interacting with these vectors is intentionally similar to using plain Scheme vectors:

```
> (define v (make-cvector _int 10))
> (cvector-length v)
10
> (cvector-set! v 5 55)
> (cvector-set! v 15 55)
cvector-ref: bad index 15 for cvector bounds of 0..9
> (cvector-ref v 5)
55
```

These vectors can be used as inputs to foreign functions via the _cvector type.

SRFI 4 [6] defines similar structures, except that there are different Scheme types (therefore different function names) for each kind of vector, making it limited to numeric vectors. Our 'foreign' interface adds a complete re-implementation of SRFI 4, which will replace the C-based module that is currently a part of PLT Scheme[6].

3.2.4 Struct Types

The C part of our implementation provides limited support for defining struct types: we get a 'makecstructtype' function which constructs a new kind of primitive type given a list of existing types. This new type can be used with Scheme pointer objects, which will cause copying the structure *contents* rather than the pointer value when marshaling data. Accessing these objects is left for the Scheme side, which uses the information given by the ctype-sizeof and the ctypealignof functions to compute the offsets into the contained values.

This functionality is sufficient for the 'foreign' module to make C structs accessible from Scheme. Two interfaces are provided:

1. _list-struct is a type constructor: given a list of type objects, it constructs a matching C struct type, and wraps the result in a yet another type that translates values contained in such a C struct value to and from a Scheme list of values. Using this type is simple, but it involves extra allocations which is an extra overhead some users will want to avoid.
2. definecstruct is a new syntax, similar to PLT Scheme's 'definestruct', except that slots have an associated type. Values of this new type are kept as a pointer object that references the memory block holding the binary data. Again, this simplifies interfaces: there is no overhead involved as we are dealing with the raw data. A simple example of using such a struct type follows:

```
> (define-cstruct _foo
    ((x _int) (y _double)))
> (define x (make-foo 1 2.3))
> (foo? x)
#t
> (list (foo-x x) (foo-y x))
(1 2.3)
> (set-foo-y! x 4.5)
```

3.2.5 Simple Function Types

Finally, the core functionality that allows interactions with foreign libraries is enabled by the _cprocedure type constructor. This constructor creates a function type when given a list of input types and an output type. Like all other C type objects, the resulting function type has two translation procedures: one going from C to Scheme and one going back. For these function types, the first translator generates a *callout* object that can be used as a new Scheme primitive, and the second generates a callback object that can be sent to C code allowing it to invoke a Scheme procedure. This internal function is implemented via the primitive 'fficall' and 'ffi-callback' functions (see Table 2), it's definition is (roughly[7]):

```
(define (_cprocedure itypes otype)
  (make-ctype _pointer
    (lambda (x) (ffi-callback x itypes otype))
    (lambda (x) (ffi-call x itypes otype))))
```

This means that from the user's point of view, a simple type specification like '(_cprocedure (list _int _int) _int)' can be used as either an input or an output type, and it can properly nest (negative function type occurrences generate callbacks and positive occurrences generate callouts). For example, the following contrived higher-order C function:

```
int foo_ho_ho_func(int x, int(*(*f)(int))(int)) {
  return (f(x+1))(x-1);
}
```

can be used (interactively!) in Scheme in a straightforward way:

```
> ((get-ffi-obj "foo_ho_ho_func" "foo.so"
     (_cprocedure
       (list _int
             (_cprocedure
               (list _int)
               (_cprocedure (list _int) _int)))
       _int))
   3
   (lambda (x) (lambda (y) (+ y (* x x)))))
18
```

3.2.6 Complex Function Types: IDL Features

The _cprocedure can generate simple interfaces, but it is insufficient in cases where the foreign function needs an additional layer of interface when arguments and/or the return value on the Scheme side don't match those of the foreign side. A common example of

[6] The current implementation does not deal with the external syntax specified in SRFI 4.

[7] The actual implementation accepts another optional argument that can be used to tweak the resulting primitive procedure. This is described in the following section.

this is a foreign function that expects a pointer and a size indicator, which correspond to a single Scheme object that encapsulates both. For example, the standard C 'read' function expects a string buffer and its size in two input arguments. A simple _cprocedure-generated interface inevitably exposes the additional argument, so the interface programmer needs to wrap it by additional glue code. For this, _cprocedure has an extra optional argument that is expected to be a procedure that wraps the resulting foreign function[8]:

```
(define c-read
  (get-ffi-obj "read" "libc.so.6"
    (_cprocedure (list _int _string _int) _int
      (lambda (prim)
        (lambda (fd buf)
          (prim fd buf (string-length buf)))))))
```

Another common example is the use of 'output pointers' by foreign code to return multiple values. Again, a naive _cprocedure interface will be awkward to use from Scheme code, and the interface programmer needs to use a wrapper that makes the foreign function more Scheme-friendly:

```
(define c-modf
  (get-ffi-obj "modf" "libc.so.6"
    (_cprocedure (list _double _pointer) _double
      (lambda (prim)
        (lambda (d)
          (let* ([p (malloc _double)]
                 [r (prim d p)])
            (values (ptr-ref p _double) r)))))))
```

More forms of wrappers are needed in other situations: additional argument dependencies, input- and output-pointers, different allocation strategies, implicit 'self' pointers, etc. In general, we need a way to combine arbitrary wrappers that operate on arbitrary arguments. Such wrappers cannot be implemented as new C types, since such types can add layers of processing on each value independently, rather than the required interaction among multiple arguments and output values. What we need here is some form of an interface description language (IDL). The requirements for an appropriate IDL are:

- it should be easy to write and easy to read,
- it should be rich enough to express interactions such as the two demonstrated above as well as others,
- it should not lead to an expensive performance hit,
- it should be easy to extend when facing new situations.

One way that we have tried to tackle this issue is by providing the necessary abstractions as a collection of procedures, each performing a single task, and have interfaces use combinators to build the required argument interactions. This approach has a major drawback: it leads to complex expressions which are hard to write and harder to read. Using this approach, code that converts a Scheme string argument to buffer-size and pointer arguments might use a 'string+len' function together with combinators that arrange to swap the arguments, for example:

```
(define foo
  (get-ffi-obj "foo" "foo.so"
    (_cprocedure (list _int _string) _int
      (compose prim:1+2->2+1
               (prim:1->1+2 string+len)))))
```

[8]Actually, our interface is part of the new version of PLT Scheme, which has a new *byte-string* type for raw (non-Unicode) character sequences. We use strings in the following examples for simplicity.

It is obvious that this code is hard to read — for example, inspecting the types reveals that there is a bug in this code[9]. In addition, such procedures will often be higher order for customization, making things even worse. Another drawback of this approach is the number of procedure applications that are involved in each call: any time overhead involved in foreign calls might be critical, and we don't want programmers to move to inferior tools because of it.

The approach that our system takes uses Scheme's syntax abstraction capabilities instead. We define a new type combinator, _fun, which is actually a syntax transformer. Usages of _fun generate the appropriate wrapper code, and use _cprocedure with it to create the function type.

Simple usages of _fun are similar to _cprocedure except that the types need not be put in a list, and an infix '->' marker separates the input types from the output type. For example, using _fun for the higher-order C example from Section 3.2.5:

```
> ((get-ffi-obj "foo_ho_ho_func" "foo.so"
     (_fun _int (_fun _int -> (_fun _int -> _int))
           -> _int))
   3
   (lambda (x) (lambda (y) (+ y (* x x)))))
18
```

In its simple form, the _fun type constructor has this syntax:

(_fun ⟨*f-type*⟩* -> ⟨*f-type*⟩)

which covers simple function interfaces in a slightly more convenient form than _cpointer. In its full form, _fun is extended to deal with common argument interactions like most IDLs and more — rather than fighting with a limited preprocessor or re-implementing a C parser, we have a real (meta) language to help us. Using syntactic abstractions in Scheme, we achieve a powerful IDL through _fun, one that can be extended to handle all possible situations.

The full form of the _fun syntax has two optional parts, and each ⟨*f-type*⟩ subform can have an optional identifier and/or expression:

(_fun [⟨*args*⟩ ::] ⟨*f-type*⟩* -> ⟨*f-type*⟩ [-> ⟨*expr*⟩])

⟨*f-type*⟩ ::= ⟨*t-expr*⟩ | ([⟨*id*⟩ :] ⟨*t-expr*⟩ [= ⟨*expr*⟩])

⟨*t-expr*⟩ ::= *expressions that evaluate to a type value*

The sequence of ⟨*f-type*⟩s in their full form behave like a sequence of 'let*'-bindings, each with an associated type and a value (both plain Scheme expressions). As with 'let*', value expression can refer to previous identifiers for their values. Omitting an identifier makes the corresponding value inaccessible for subsequent expressions; omitting a value expression means that the resulting wrapper function will expect a corresponding argument. For example, in this definition:

```
(define c-read
  (get-ffi-obj "read" "libc.so.6"
    (_fun _int
          (buf : _string)
          (_int = (string-length buf))
          -> _int)))
```

there are three arguments that are passed on to the foreign function:

- The first uses the short form: it has no value so it will receive the first value passed on to 'c-read', and it has no name so its value can not be used in following expressions.
- The second argument has no value too, making it get the sec-

[9]A type checker will help avoiding such errors, but will not make things easier to read and write.

ond '`c-read`' argument, and its value is bound to '`buf`'.

- The third argument has a value expression so the value that is passed on to the foreign function is always the length of the second (string) argument.

'`c-read`' is therefore a Scheme procedure that expects two arguments and returns an integer, by arranging for properly calling the foreign '`read`'.

In some rare cases, an interface needs to have better control of the wrapper's argument list — which is the purpose of the optional '⟨*args*⟩ `::`' prefix: it specifies the arguments to the resulting wrapper function. For example, if '`read`' were to expect the buffer size first, we would use this `_fun` type:

```
(_fun (fd buf) ::
      (fd : _int)
      (_int = (string-length buf))
      (buf : _string)
      -> _int)
```

Note that identifiers are important here, as they connect the foreign inputs with the wrapper's inputs. The ⟨*args*⟩ part can also be used to specify normal Scheme argument lists, including optional arguments.

A second '`->`' marker denotes a result expression different than the one that the foreign function returned. This expression can use any bound values and arguments, as well as the foreign result value (if given an identifier). For example, the '`modf`' interface given above is better written with `_fun` as:

```
(define c-modf
  (get-ffi-obj "modf" "libc.so.6"
    (_fun _double (p : _pointer = (malloc _double))
          -> (r : _double)
          -> (values (ptr-ref p _double) r))))
```

The fact that we can insert any Scheme expression for the return value makes it easy to change such definitions so they use alternative ways for assembling the return values, for example, changing '`values`' to '`cons`' in the above. If this was implemented in C, such changes would require more work.

The similarity between the `_fun` syntax and '`let*`' is not incidental: `_fun` assembles a wrapper function that contains a single '`let*`' expression, which evaluates the various expressions, binding the results to specified identifiers. For example, the usage of `_fun` in the last example expands to:

```
(_cprocedure (list _double _pointer) _double
  (lambda (ffi)
    (lambda (tmp15)
      (let* ((p (malloc _double))
             (r (ffi tmp15 p)))
        (values (ptr-ref p _double) r)))))
```

This satisfies the efficiency requirement: only one extra function call is wrapped around the foreign call.

3.2.7 Additional IDL Features: Custom Function Types

The `_fun` facility handles some common cases where we need to bridge a gap between the foreign function and Scheme code that uses it, but there are additional cases that are not addressed. For example, the '`modf`' interface code above represents such a common situation – output pointers that are used by foreign code to return multiple values. We therefore extend the `_fun` syntax further, by making it interact with special 'custom function types' that are themselves syntaxes — such types can install pieces of code that are used before and after the foreign call, possibly modifying the corresponding value. In the case of output pointers we want to allocate some memory before the foreign call and dereference it afterward, a task that is achieved by the `_ptr` custom type. `_ptr` is a syntax with usages that has the following form:

Custom Type	Usage
`_ptr`	input, output, or input/output pointers
`_box`	similar to an input/output `_ptr`, but modifies the Scheme box contents (PLT Scheme has a mutable box type. Note that we don't need to associate Scheme boxes with 'shadow' pointers: either copy values, or use a `_pointer` instead of a box)
`_list`, `_vector`	marshal lists and vectors as C pointers
`_bytes`	uses Scheme byte-strings (raw, non-Unicode strings)
`_?`	a special non-type intended for saving intermediate interface results

Table 4. Simple types defined by the Scheme module

(`_ptr` ⟨*mode*⟩ ⟨*type-expr*⟩)

⟨*mode*⟩ ::= `i` | `o` | `io`

The ⟨*mode*⟩ specifies an input, output, or input/output pointer. In the '`modf`' case, we use an output pointer:

```
(define c-modf
  (get-ffi-obj "modf" "libc.so.6"
    (_fun _double (p : (_ptr o _double))
          -> (r : _double) -> (values p r))))
```

The code that is generated by this `_fun` syntax is similar to the previous code,

```
(lambda (tmp15)
  (let* ((p (malloc _double))
         (r (ffi tmp15 p))
         (p (ptr-ref p _double)))
    (values p r)))
```

but notice that we don't need to explicitly allocate a double or dereference the pointer.

The custom function types that are provided by the '`foreign`' are listed in Table 4. Further details on these types can be found in our user manual.

As mentioned above, Custom types are implemented as syntaxes. `_fun` tries to expand each type expression it encounters, and if an expansion is identified as a custom type, then it has certain forms that contain the relevant pieces of code. A custom type expansion is a '(⟨*key:*⟩ ⟨*val*⟩ ...)' sequence where all of the ⟨*key:*⟩s are from a short list of known keys. Each key interacts with generated wrapper functions in a different way, which affects how its corresponding argument is treated:

`type:` specifies the foreign type to be used (`#f` can be used to make this not participate in the foreign call).

`expr:` specifies an expression to be used for arguments of this type, removing it from wrapper arguments.

`bind:` specifies a name that is bound to the original argument if it is required later (e.g., `_box` needs to refer to the original box).

`1st-arg:` specifies a name that can be used to refer to the first argument of the foreign call (good for common cases where the first argument has a special meaning, e.g., for method calls).

`prev-arg:` similar to `1st-arg:`, but refers to the previous argument.

pre: a pre-foreign code chunk that is used to change the argument's value.

post: a similar post-foreign code chunk.

The following is the implementation of the _ptr custom type from the 'foreign' module. It is provided to roughly demonstrate how this is done; again, complete details are given in the user manual.

```
(define-syntax _ptr
  (syntax-rules (i o io)
    [(_ i  t)
     ;; input: malloc a pointer, set its value from the argument
     (type: _pointer
      pre:  (x => (let ([p (malloc t)]) (ptr-set! p t x) p)))]
    [(_ o  t)
     ;; output: malloc a pointer on entry, dereference on exit
     (type: _pointer
      pre:  (malloc t)
      post: (x => (ptr-ref x t)))]
    [(_ io t)
     ;; input/output: like output, but set its contents on entry
     (type: _pointer
      pre:  (x => (let ([p (malloc t)]) (ptr-set! p t x) p))
      post: (x => (ptr-ref x t)))]))
```

All of the special custom types provided by 'foreign' are defined this way.

To conclude: our _fun satisfies all requirements mentioned above for a good IDL: it is easy to read and write, it can express all wrapper interactions that other IDLs can express and more, it is efficient, and extensible by the ability to add new custom types that handle new kinds of processing. As expected from a syntax transformer that performs some substantial work, it carries some conceptual overhead, but we believe that overall it is better than the C processing alternatives since Scheme is superior in its syntactical abstraction capabilities.

4 Usage Examples

With the implementation of our system, we provide a few (mostly Linux) library interfaces. This was used to test the implementation, motivating the overall design. We now describe a few examples of using our system, all based on these interface implementations.

Syntactic Abstractions

C provides some (limited) degree of syntactic abstraction, whereas Scheme truly shines in this area. When a complete library interface is desired (rather than pulling out a few useful functions), repetition is common. Writing interfaces in Scheme makes such problems almost non-existent — for example, our ImageMagick interface uses a simple macro:

```
(define-syntax defmagick
  (syntax-rules (:)
    [(_ id : x ...)
     (define id
       (get-ffi-obj 'id libwand (_fun x ...)))]))
```

to make interface definitions easier.

Defining new syntaxes can help in other, less common situations. For example, KSM [4] has a `clang:sym` form that exposes a foreign library variable as a Scheme binding. Using PLT Scheme macros, we can achieve this functionality in Scheme using a macro that defines the C 'variable' as a macro[10]:

```
(define-syntax defcvar
  (syntax-rules ()
    [(_ var lib type)
     (define-syntax var
       (syntax-id-rules (set!)
         [(set! var1 val1)
          (set-ffi-obj! 'var lib type val1)]
         [(var . xs)
          ((get-ffi-obj 'var lib type) . xs)]
         [var (get-ffi-obj 'var lib type)]))]))
```

and verify that it is working properly:

```
> (defcvar z "x.so" _int)
> z
0
> (set! z 123)
> z
123
> ((get-ffi-obj "getz" "x.so" (_fun -> _int)))
123
```

where the C code that is compiled into "x.so" is:

```
int z = 0;
int getz() { return z; }
```

Using Types

C types in our system are somewhat lighter than expected: there is only a loose correlation between these types and Scheme object types. A type in our context can simply mean a different way of marshaling Scheme values to/from C, for example, the _file type from Section 3.2.1 is simply a different way to marshal MzScheme path objects which are normally used with _path. No C-level support is needed for such cases: there are no new binary tags involved, and no new object representations at the implementation level, meaning that it is extremely cheap to create such type descriptors. A common usage of types is therefore as a simple mechanism to add hook on the translation process.

For example, the ImageMagick library specifies a 'MagickWand' type, which is always being manipulated as a 'MagickWand*' pointer. There are functions that return a pointer to a newly created 'MagickWand' object, and these objects must be destroyed with the 'DestroyMagickWand' function. To do this automatically, we define a _MagickWand type using _pointer and providing a new translation when going from C to Scheme, one that uses 'registerfinalizer' to make the GC use 'DestroyMagickWand' when reclaiming the pointer object[11]:

```
(define _MagickWand
  (make-ctype _pointer
    #f ; Scheme->C translation is the same as _cpointer
    (lambda (ptr)
      (if ptr
        (begin (register-finalizer ptr destructor) ptr)
        (error '_MagickWand "got a NULL pointer")))))
```

We can make this even better with a new cpointer type which uses an appropriate tag to identify these pointers and make sure that we don't confuse pointers to internal ImageMagick objects of different types. The following definition uses 'definecpointertype' (see Section 3.2.2) to create a type that tags all pointers when they are

[10] From the MzScheme [9, Section 12.1] manual: The 'syntax-idrules' form has the same syntax as 'syntaxrules', except that each pattern is used in its entirety (instead of starting with a keyword placeholder that is ignored).

[11] This assumes that there is no way to get a second pointer object that refers to the same 'MagickWand' object, so care should be taken with functions that can create such aliases.

moved from the foreign side to Scheme, and check the tag when sending a Scheme pointer object out to foreign code.

```
(define-cpointer-type _MagickWand #f #f[12]
  (lambda (ptr)
    (if ptr
      (begin (register-finalizer ptr destructor) ptr)
      (error '_MagickWand "got a NULL pointer"))))
```

A different example of using a new type comes from our TCL interface: the `Tcl_Eval` function returns a status integer, indicating a possible error. In our implementation, we define `evaltcl` as:

```
(define eval-tcl
  (get-ffi-obj "Tcl_Eval" libtcl
    (_fun (interp : _interp = (current-interp))
          (expr : _string)
          -> _tclret)))
```

using the following `_tclret` definition:

```
(define _tclret
  (make-ctype (_enum '(ok error return ...))
    (lambda (x) (error "tclret: only for returning"))
    (lambda (x)
      (when (eq? x 'error)
        (error 'tcl (get-string-result
                     (current-interp))))
      x)))
```

which effectively translates a TCL error into a Scheme exception.

Note that the TCL interface uses a Scheme parameter '`current-interp`' as the value of the first argument to '`TCL_Eval`'. We can make this implicit by defining a new custom type syntax, using the '`expr:`' keyword:

```
(define-syntax _cur-interp
  (syntax-id-rules ()
    [_ (type: _interp expr: (current-interp))]))
(define eval-tcl
  (get-ffi-obj "Tcl_Eval" libtcl
    (_fun _cur-interp (expr : _string) -> _tclret)))
```

Using Custom Types

Custom types are intended to be used in situations where simple independent processing of each argument is insufficient. For example, many functions in the ImageMagick interface return a 'status' integer that indicates if there was an error. If an error has occurred, the main object involved in the function invocation should be used to retrieve the error message and severity. One way to deal with this situation is to save the object in a place accessible right after the foreign call, like a parameter. This is essentially what the TCL interface does, where `_tclret` uses a parameter to get the error message. The ImageMagick interface is different — instead of a single implicit context parameter, it fits more an object-oriented style, where each method call happens in its object's context.

As a result, a good interface must be able to provide a relation between different arguments, namely the result value (to be checked for an error) and the first argument (providing the current object context). This is done using the `1st-arg:` keyword of a custom type which specifies an identifier that will be bound to the first argument:

```
(define-syntax _status
  (syntax-id-rules (_status)
    [_status
     (type: _bool
      1st-arg: 1st
      post: (r => (unless r
                    (raise-wand-exception 1st))))]))
```

Memory Management

Usually, there are important aspects of the library interface that are not fully specified. Memory management issues often fall under this category. For example, a naive interface might behave in a surprising way:

```
> (define crypt
    (get-ffi-obj "crypt" "libcrypt"
      (_fun _string _string -> _string)))
> (define a (crypt "foo1" "23"))
> a
"23.kLNfMwUW0Q"
> (define b (crypt "foo4" "56"))
> b
"568.5HohJYC0g"
> a                    ; a is modified!
"568.5HohJYC0g"
> (string-set! a 0 #\X) ; verify that a and b
> (list a b)           ; are the same string
("X68.5HohJYC0g" "X68.5HohJYC0g")
> (eq? a b)            ; ...but not quite the same
#f
```

Using a simple SWIG interface, made using the C prototype declaration for '`crypt`':

```
extern char *crypt(const char *key, const char *salt);
```

suffers from this problem too. The reason for this strange behavior is that both our interface implementation and SWIG's generated code use MzScheme's '`make_string_without_copying`' function, which simply wraps an existing C string in a Scheme string object. The standard Unix `crypt` function returns a pointer to its own static string, making the above interaction create two Scheme string objects that point to this static buffer — but the Scheme objects are still different. This can be dangerous as it breaks an implementation assumption, so some solution is required. Changing the implementation to use '`scheme_make_string`' would not be acceptable in the general case since it leads to an expensive overhead. In addition, there are other foreign functions (e.g., `getcwd`) that can allocate a return string, and blindly copying it will cause a memory leak (the allocated string is not in GC-controlled memory).

Using our system simplifies such a solution since we don't have to break out of Scheme, we can simply use a new type[13]:

```
(define _string/copy
  (make-ctype _string #f
    (lambda (x) (string-append x #""))))
```

We can solve numerous problems in a similar way, for example, using semaphores to avoid problems with the single crypt buffer, or creating a new `_string/free` that copies a string and freeing the previous GC-invisible one.

[12] Use `_cpointer` as a base type, no extra translation when going to from Scheme to C, and register the destructor on the way back.

[13] Note that this is not relevant now, since our system is part of the Unicode-enabled MzScheme, so Scheme strings are stored in Unicode format, meaning that they are always copied.

5 Related Work

The first and foremost advantage that our foreign interface has over existing implementations, is the fact that it is truly dynamic. This means that functionality that traditionally is available only via C code is available to Scheme programmers, which makes for a compiler- and architecture-independent system. Furthermore, the dynamic aspect of the system allows for playing with foreign extensions dynamically, modifying and debugging the interface at runtime[14]. Exploratory programming is therefore possible, hence the overall development cycle becomes much lighter.

A second advantage comes from the fact that we use Scheme. Using a language with robust syntactical abstractions makes it possible to provide an IDL-like interface for interface programmers, with features that can go beyond capabilities of conventional IDLs [18, 16]. Having syntactic abstractions in the language makes it possible for users to extend their own code using new constructs, including ones that are unique to a single library, in contrast to fixed IDLs that are either fixed, or used through a primitive facility like the C preprocessor.

Dynamic interfaces are not as common as static interfaces. Existing dynamic systems, for example the Allegro CL foreign function interface [10] and Python's ctype module, do not provide the low-level C-substitute features that we do. Urban's FFI survey [17], although a little out-dated, provides an excellent overview on existing systems and implementation issues. It is interesting to note an SML interface system [2] as another, somewhat similar system to ours. Similar to our design, the main idea is *data-level interoperability* [8] — making raw C data available to the high-level language, but our system differs in a few important aspects:

- Our design is built around the idea of enabling arbitrary C-like unsafe code — whereas Blume's system uses SML's type system to enhance interaction with foreign code.
- Our system goes one step further in giving users more power. "If you can do it in C, then we will let you do it in Scheme" rather than "Some C-level operations are useful enough that we let you use them".
- Blume's system is limited to SML's syntactic framework, where we use Scheme's capabilities for creating IDL-like syntax.

We focus our comparison on static interface generators such as GreenCard [14], G-Wrap [3], and SWIG [1]. There are Scheme systems that fall under this category too by providing support for combining Scheme and C code, for example, Gambit-C[15] [5] and KSM [4]. Most notably, SRFI 50 [15] attempts to standardize this approach, possibly making it possible for different Scheme implementations to share C code. These systems make it possible to write Scheme code that is converted to C code, so it is easy to write such 'Scheme' code that calls C functions as if they were plain function calls. Some of these systems lack a code generation component that is derived by an IDL or some equivalent, but they can all be seen as static code generators.

We now focus on SWIG as a popular system that can be used for multiple high-level languages. A simple translation using SWIG requires the user to compile (through the SWIG parser) a C header file with a SWIG interface file, resulting in C code that is then, yet again, compiled using a C compiler, to produce a C module that is finally imported into Scheme. In contrast to the static approach, our '`foreign`' library makes it possible for a Scheme developer to quickly open up a C library, pull out a few procedure objects and start an interactive development session.

[14] As long as no fatal errors occur.

[15] Some parts of this were ported to PLT's MzC compiler.

func.	Glue Type	CPU	Real	GC
`crypt`	SWIG	38%	4%	-34%
	Handwritten C glue	53%	49%	0%
`sqadd`	SWIG	55%	57%	0%
	Handwritten C glue	60%	61%	0%

Table 5. Comparison of overhead time

It could be argued that a simpler, more user friendly system comes at a price of expensive overhead, leading to an inherent sacrifice of performance. Testing out two simple benchmarks, we found that the interface overhead of our system is just slightly slower as a compiled interface that was generated by SWIG, which itself has an almost identical overhead to hand-written glue code.

Our results are summarized in Table 5. Two functions were used for this analysis — the first is the `crypt` function taken from the standard Unix `libcrypt`: consuming two strings and producing an encrypted string result. The second is a simple C function, `sqadd`, that performs an addition of two integer squares. We measured a million executions of `crypt` and 30 million executions of `sqadd`, performing each test for 16 rounds beginning with a fresh MzScheme process, discarding the 6 extreme timings and averaging the other 10. The percentages are computed as: $\frac{Time_{PLT}-Time_{RawC}}{Time_{SWIG}-Time_{RawC}}-1$ where $Time_{PLT}$ is the averaged running time of our interface, $Time_{SWIG}$ is the average running time of SWIG, and $Time_{RawC}$ is that of an implementation of comparable repetition loops in C. The same computation was used to compare our system against handwritten C glue code.

As Table 5 shows, our system is about 1.5 times slower then SWIG, and, in most cases the handwritten glue code. The biggest performance hit is in the simple arithmetic function, where the actual foreign code does much less than the interface code. Situations like this should rarely occur since the usual case of using a foreign library is when it can do some substantial work that is otherwise hard to achieve in Scheme.

While issues of timing and performance are important, aspects such as implementation complexity and ease of use must also be considered. Comparing our system to SWIG and interfaces that use an IDL, it becomes clear that our implementation is better in at least one aspect. One advantage that our system provides over the static approaches is the ability to specify additional functionality using new user-defined types that involve arbitrary translation code. The main point here is that such translations are written in the high-level language itself rather than dealing with the intricacies of the C implementation.

In addition, regardless of interface design and syntactical complexity, our implementation is better because the interfacing mechanism itself is in a high order language: making it possible to include arbitrary Scheme code as part of the foreign call specification. This is further enhanced by the fact that we use Scheme since it is possible to create new syntactic abstractions to deal with new requirements. Either with SWIG interface files or with an IDL, the interface developer is still confined by C and C-like code with its known shortcomings when it comes to dealing with complex problems.

6 Future Work

C++ Libraries Currently, there is support only for plain C libraries. Depending on implementation details, it can be feasible to interface C++ libraries. This might involve plenty of details regarding object layout, inheritance, virtual function tables, name mangling, etc. Hopefully, these issues can be addressed in Scheme so we might not need any further enhancements to the C part of our implementation.

Parsing C One of the main disadvantage of our system is that it is not using C, so we cannot use C include files as rough interface specifications. We plan to investigate a simple C header-file parser that will parse files into s-expressions, which can be used to automate some aspects of interface generation (A working parser prototype exists). Such a parser does not need to be fast and efficient, since parsing can be done at syntax expansion time, eliminating any run-time speed costs. In addition, note that as usual with other interface generators, this will almost never mean that an interface can be fully automated, as header files do not provide enough information — this situation might improve if we target some IDL language instead (most use similar syntax).

Memory Management Issues Currently, our system works well with both versions of PLT Scheme: the one that uses the Bohm conservative garbage collector and the one that uses a precise moving collector. However, there are still issues that interface writers need to be aware of. In time, we will gain more experience writing interfaces, which will motivate further functionality that will make this easier — our goal is, of course, making GC-related issues as transparent as possible for interface writers.

One aspect of this, is dealing with struct objects that might contain GC-able pointers. We have a plan to deal with this, effectively making it possible to specify in Scheme a map of pointer offsets that the garbage collector should be aware of, making it treat new Scheme-defined structs properly.

Additional Scheme Support There are some areas in which additional Scheme support is needed. For example, an array of structs is hard to deal with — there is no way to get to one such struct and modify it, since accessing it will create a copy. We believe that it is possible to write Scheme code that will make this possible, by not pulling out a struct copy, but rather provide forms that will use nested reference indexes, where some are vector indexes and some are struct field names. If we can make this composable, it would be possible to deal with them in an easy way — without resorting to pointer aliasing[16].

An additional area where additional support is needed, is when dealing with foreign functions that block. MzScheme contains a few hooks that are intended to be used when it is embedded as a library, these hooks can be used for calling blocking foreign functions as well.

Using Contracts PLT Scheme has support for procedure contracts [7] which could be used to enhance the robustness of library interfaces. Specifically, we want to treat contract violations in modules that use the '`foreign`' module as more severe, as these are equivalents of C bugs, which might result in a crash. A module would also need some way of declaring it as a proper interface, meaning that code that uses it should not be blamed for crashes. Alternatively, code that is not intended as an interface (i.e., code that provides functionality for interface modules) should propagate the property of contract violation severity.

[16]The precise garbage collector makes it impossible to get a pointer to the internal part of an allocated block

Assembly Code Generation Working our way to native just-in-time compilation, we plan on adding machine-code generation ability to PLT Scheme. We will interface this functionality via the '`foreign`' module. Furthermore, some of the interface aspects can be implemented in assembly when runtime is important.

7 References

[1] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, July 1996.

[2] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL'01: First workshop on multi-language infrastructure and interoperability*, September 2001.

[3] Rob Browning. G-Wrap home page. `http://www.nongnu.org/g-wrap/`.

[4] Hangil Chang. KSM-Scheme home page. `http://square.umin.ac.jp/ hchang/ksm/`.

[5] Marc Feeley. Gambit Scheme system. `http://www.iro.umontreal.ca/ gambit/`.

[6] Marc Feeley. SRFI 4: Homogeneous numeric vector datatypes. `http://srfi.schemers.org/srfi-4/`.

[7] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[8] Kathleen Fisher, Riccardo Pucella, and John Reppy. Data-level interoperability. Bell Labs Technical Memorandum, April 2000.

[9] Matthew Flatt. *PLT MzScheme: Language Manual*. PLT, August 2004. Version 208.

[10] Franz Lisp. Foreign function interface. `http://www.franz.com/-support/documentation/6.1/doc/foreign-functions.htm`.

[11] Anthony Green. The libffi home page. `http://sources.redhat.com/libffi/`.

[12] Samuel P. Harbison. *Modula-3*. Prentice-Hall, 1992.

[13] Thomas Heller. The ctypes module. `http://python.net/crew/theller/ctypes/`.

[14] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. GreenCard: a foreign-language interface for Haskell. In J. Launchbury, editor, *2nd Haskell Workshop*, 1997.

[15] RIchard Kelsey and Michael Sperber. SRFI 50: Mixing scheme and c. `http://srfi.schemers.org/srfi-50/`.

[16] The Open Group. *CAE Specification, DCE 1.1: Remote Procedure Call*, chapter 4. The Open Group, October 1997.

[17] Reini Urban. Design issues for foreign function interfaces. `http://xarch.tu-graz.ac.at/autocad/lisp/ffis.html`, Last updated at 2004.

[18] A. Vogel, B. Gray, and K. Duddy. Understanding any IDL — lesson one: DCE and CORBA. In *Proceedings of the Third International Workshop on Services in Distributed and Networked Environments (SDNE'96)*, 1996.

Acknowledgments

We would like to thank Matthew Flatt: this work would not be possible without his help, especially with GC-related issues. The comments and suggestions made by the reviewers have been extremely helpful, Mike Sperber was particularly helpful in the process of revising this text.

Debugging Scheme Fair Threads

Damien Ciabrini
INRIA Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
Damien.Ciabrini@sophia.inria.fr

Abstract

There are two main policies for scheduling thread-based concurrent programs: preemptive scheduling and cooperative scheduling. The former is known to be difficult to debug, because it is usually non-deterministic and can lead to data races or difficult thread synchronization. We believe the latter is a better model when it comes to debugging programs.

In this paper, we discuss the debugging of Scheme Fair Threads, that are based on cooperative scheduling and synchronous reactive programming. In this approach, thread communication and synchronization is achieved by means of special primitives called signals, which ease the debugging process. We present the tools we have implemented to deal with the main types of concurrent bugs that can arise in this special programming framework.

1 Introduction

Modern systems offer multitasking inside a single application: there can be many virtually independent flows of control, usually called *threads*. These are commonly used in programs nowadays.

Concurrent programming is a difficult task. First, because reasoning about interleaved flows of control is an intrinsically difficult task. Second, because bugs caused by multi-threaded programming are usually very difficult to track down with traditional debuggers.

There are various policies for scheduling multi-threaded programs. The two major categories are *preemptive* scheduling and *cooperative* scheduling. Each one comes with its pros and cons with respect to debugging.

1.1 Preemptive or Cooperative Scheduling

Preemptive scheduling appeared in operating systems [13] in the late 70s and has been democratized in languages in the mid 90s. In this model, the thread library (usually the underlying OS) may suspend the execution of a thread at any time to schedule another one. It can also benefit from Symmetric Multi-Processor hardware (henceforth SMP). Unfortunately, preemptive multi-threading is implemented in a way that leads to non-deterministic scheduling. It is known to be difficult to program with and painful to debug:

- Locks have to be acquired before accessing shared memory to avoid data races. Omitting locks may cause data corruption, in which case debuggers become almost useless.
- Synchronization by means of *mutexes* can be missed if notifications are sent before some threads started to await them. In this case, debuggers hardly help because they do not provide tools for tracing the order of synchronization.
- It is very difficult to reproduce a bug because one cannot play the same execution twice. Actually, the simple fact of inserting `prints` in a program is sufficient to make a bug no longer appear at run-time.
- Complex features like priority boost or scheduling policies are non-portable, and debuggers usually do not provide support for them. Using these features can lead to bugs like priority inversion [20], that are difficult to track or to explain.

Cooperative scheduling is an older model, in which it is the responsibility of threads themselves to *cooperate*, *i.e*, to give back control so that another thread can continue to execute. This is a deterministic model where only one thread is executing at a time (which hardly benefits from SMP). This scheduling model greatly eases the debugging for various reasons:

- Debuggers do not have to deal with data races, since only one thread is active at a time.
- The scheduler is deterministic. This means that when a bug occured, it can be easily reproduced by replaying the same execution.

In cooperative schedulers, problems like dead-locks can still occur. Moreover, some specific problems are introduced because of manual cooperation:

- If a thread fails to cooperate, the whole program is blocked.
- Too few cooperations can lead to interactivity problems, for instance in Graphical User Interfaces (henceforth GUI).
- Too many cooperations can lead to unnecessary context switches and poor performance. This is a problem similar as taking too many locks to protect shared variables.

Contrary to bugs caused by non-determinism, these types of bug are much easier to detect and to correct with the help of a debugger.

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

1.2 An Hybrid Solution

The Bigloo [18] Scheme compiler provides an alternative model for multi-threaded programming called Scheme Fair Threads [17]. It is a programming framework based on *synchronous reactive programming*[1] with the following characteristics:

- It provides a cooperative and deterministic scheduler. Thus there is no need to acquire locks and execution can be replayed at will.
- Threads communicate by broadcasting *signals* into the scheduler. It is guaranteed that signals are seen by all threads during a logical round of schedule called an *instant*.
- It still provides the ability to do I/O operations asynchronously, *i.e*, without blocking the scheduler and also by taking advantage of SMP. Non-determinism due to asynchronous I/Os is confined into well defined locations in the scheduler.

In Scheme Fair Threads, a debugger still has to deal with dead-locks or live-locks, as it is the case with classical cooperative or preemptive scheduling. However, it does handle communication and synchronization bugs differently, because mutexes are abandoned in favor of broadcast signals. The debugger has to provide a new set of tools to deal with specific problems introduced by this programming framework.

We have extended BUGLOO [2], a source-level debugger for Bigloo programs, in order to support the debugging of fair threads. In this paper, we do not talk about POSIX-like mutexes or condition variables, as Fair Threads do not use them to communicate. Instead, we concentrate on the debugging of cooperation points, signals and instants:

- We have enhanced the single stepping by introducing new step points that take into account cooperative scheduling and communication by broadcast signals.
- We provide a tool for inspecting the state of fair threads and for viewing signals present in the scheduler when execution is suspended. This tool is used to fix bugs like dead-locks and live-locks that occur *during* a single instant.
- We provide a tool to graphically trace the scheduling of fair threads and the broadcasting of signals throughout the execution. It is an effective way to fix communication bugs that occur *across* many instants.

1.3 Overview

In Section 2, we detail the Fair Threads programming model and its usage in Scheme. In Section 3, we present the main debugging support for Fair Threads, namely the single stepping and state inspection. In Section 4, we describe the tool for tracing scheduler executions. In Section 5 we describe the typical usage of our tools on a producer-consumer program with asynchronous I/Os. In Section 6 we briefly describe how the debugger is implemented and we present the overall experience we had with our tools. In Section 7 we present related work. Finally, Section 8 concludes and shows some future directions for our work.

[1] See http://www-sop.inria.fr/mimosa/rp

2 Scheme Fair Threads

Scheme Fair Threads is a thread-based concurrent programming framework based on Java Fair Threads [1]. In this section we present the Fair Threads programming framework and the concept of fair scheduling. As an example, we describe the execution of an abstract program. We then give a brief overview of the Fair Threads API, along with the type of concurrent bugs that can occur in this programming framework.

2.1 Fair Threads and Fair Scheduling

In the Fair Threads model, each fair thread is mapped to a native OS thread and has its own dynamic environment. Threads are *attached* to a cooperative scheduler, in which only one thread is executed at a time. When a thread cooperates, the scheduler gives control to another thread. Note that the scheduler is deterministic, and is itself a fair thread.

The Fair Threads model has a clear semantics that emphasizes *fair* scheduling and powerful means of synchronization. Both principles are described below:

- The scheduling of fair threads is decomposed into logical units of schedule called *instants*. During an instant, fair threads communicate together by awaiting and broadcasting *signals* into the scheduler. A signal is present until the end of instant and can be associated with a value.
- If a signal is broadcast during an instant, all the threads waiting for it are guaranteed to be notified before the end of the instant. In particular, if a fair thread waits for a signal that has already been broadcast in the instant, it is immediately notified and continues its execution.
- A signal can be broadcast many times in the same instant. Fair threads can wait until the next instant to obtain the list of all the values associated with a signal that were generated in the previous instant.
- A fair thread can be re-elected for schedule in the same instant if signals have been broadcast since its last election. An instant terminates when all the fair threads have been executed and no new signal has been broadcast.

2.2 A Simple Program

To understand how fair threads are scheduled, let us describe the execution of the following abstract program composed of three fair threads.

```
      A                B                C
--------------   -----------------   ---------------
1: await sig1    1: broadcast sig1   1: await sig1
2: await sig2    2: yield            2: broadcast sig2
3: yield         3: broadcast sig3   3: await sig3
4: await sig1
```

Instant 1: fair thread *A* gets blocked in line 1 waiting for signal `sig1` to be broadcast. Next, fair thread *B* broadcasts `sig1` in line 1, then it cooperates in line 2 to explicitly complete its execution for the instant. At this point, the scheduler re-elects fair thread *A* because signal `sig1` has been broadcast in the scheduler. This fair thread then blocks waiting for signal `sig2`. Then, fair thread *C* takes the control. It does not block on signal `sig1` because it has already been broadcast (by *B*) during the instant. It broadcasts `sig2` and blocks, waiting for signal `sig3`. At this point, the instant is not

over: the scheduler re-elects fair thread *A* because signal `sig2` is now present. Then, this fair thread explicitly cooperates in line 3. At this very point all threads are blocked and no new signal has been broadcast. This marks the end of instant 1. *Signals are reset*.

Instant 2: fair thread *A* gets blocked in line 4 waiting for signal `sig1` which has not been broadcast yet during the instant. Fair thread *B* broadcasts signal `sig3` then it terminates its execution. Fair thread *C* is awakened by signal `sig3` in line 3 and it terminates. At this point all threads are blocked, the instant *2* ends and signals are reset.

Instant 3: fair thread *A* is still blocked for the instant and for the remaining instants until somebody broadcasts `sig1`.

2.3 API Overview

The Fair Threads API has been designed to be fully compatible with the SRFI-18 by M. Feeley [5]. This document proposes an extension for multi-threaded programming in Scheme, inspired by the Posix-1 API and the Java API. In Fair Threads, abstraction like mutexes or condition variable are implemented on top of signals. The previous abstract example is implemented in Fair Threads as followed:

```
 1: (define (funA)
 2:    (thread-await! 'sig1)
 3:    (thread-await! 'sig2)
 4:    (thread-yield!)
 5:    (thread-await! 'sig1))
 6:
 7: (define (funB)
 8:    (broadcast! 'sig1)
 9:    (thread-yield!)
10:    (broadcast! 'sig3))
11:
12: (define (funC)
13:    (thread-await! 'sig1)
14:    (broadcast! 'sig2)
15:    (thread-await! 'sig3))
16:
17: (define (main args)
18:    (thread-start!
19:       (make-thread funA "fairthread A"))
20:    (thread-start!
21:       (make-thread funB "fairthread B"))
22:    (thread-start!
23:       (make-thread funC "fairthread C"))
24:    (scheduler-start!))
```

We now describe the major constructions of the Fair Thread API.

2.3.1 Basic Thread Manipulation

As shown in the previous example in line 19, a fair thread is created with the `(make-thread` *`thunk . name`*`)` procedure, which takes a *`thunk`* to execute and an optional *`name`*. A fair thread must be started with `(thread-start!` *`thread`*`)` before it can be executed by the scheduler.

Cooperation is achieved by calling the `(thread-yield!)` procedure, as shown in lines 4 and 9. One thread can terminate another thread with the `(thread-terminate!` *`thread`*`)` procedure.

Unlike many threading systems, the scheduler has to be started explicitly with `(scheduler-start!)`. When started, the scheduler runs until all its threads are completed or terminated.

2.3.2 Communication by Signals

A fair thread can broadcast a signal into the scheduler with the `(broadcast!` *`sig . value`*`)` procedure. A signal can be an arbitrary Scheme object. The broadcast can be associated with an optional *`value`* that will be received by waiting threads on awake. The default value is the symbol `#unspecified`, to indicate that no particular value is associated with the broadcast of the signal.

A fair thread can await a signal by means of the `(thread-await!` *`sig`*`)` procedure. It can also await several signals at a time with `(thread-await!*` *`sigs`*`)`. At last, a fair thread can get all the values broadcast in the instant for a particular signal by using the `(thread-get-values` *`sig`*`)` procedure. The fair thread waits until the end of the current instant, and at the next instant it is awakened with the list of broadcast values.

Mutexes and condition variables are implemented on top of signals and are not presented in detail in this paper. They are still accessible by their respective SRFI-18 procedures.

2.3.3 Asynchronous I/O and SMP

Fair threads can start special *service threads* whose purpose is to do long lasting I/O operations in the background without blocking the scheduler. Such threads are standard OS threads that benefit from SMP. No lock is needed in user space because service threads cannot execute user procedures.

On I/O termination, a signal is broadcast into the scheduler to awake the fair thread that requested the operation. Here is a subset of the service threads currently supported:

- *output*: `(make-output-signal` *`p s`*`)` spawns a service thread that writes the string *s* to the output port *p*.
- *input*: `(make-input-signal` *`p n`*`)` spawns a service thread that gets *n* characters from the input port *p*.
- *socket*: `(make-accept-signal` *`s`*`)` spawns a service thread that waits for a connection on the socket *s*.
- *process*: `(make-process-signal` *`p`*`)` spawns a service thread that forks process *p* in the background.

By definition, using asynchronous I/Os introduces a certain kind of non-determinism. However, it is not harmful because it is confined into service threads, thus it cannot cause any data corruption in user space. Moreover, the fairness of the scheduler is maintained since from a thread's point of view, being notified of an I/O termination is exactly the same thing as being awakened by a signal.

2.4 Classification of Fair Threads Bugs

We saw that with Fair Threads, communication or synchronization is always based on signals and instants instead of mutexes and condition variables. In this framework, the type of bugs that can be caused by multi-threading can be classified in two subsets:

1. Bugs that can be fixed by inspecting the state of the program in the *current instant*. An example of such bug could be a deadlock that occurs because all fair threads are awaiting signals.

It could also be a bug caused by a fair thread which is stuck in a live-lock, *i.e*, a thread that repeatedly waits for a signal that has already been broadcast, thus preventing the instant to terminate. To fix this kind of errors, we provide two tools: an enhanced single stepper and a scheduler and fair thread inspector. Both are presented in Section 3.

2. Bugs for which one needs to remember the state of the program *several instants backward* in time. For example, in a badly designed sequence of successive communications, a fair thread can await a signal in a particular instant while it was broadcast in a previous instant. Dead-locks and live-locks can also be caused by a succession of wrong synchronization. To fix this kind of errors, we provide a tool to graphically visualize what happened in the scheduler during a succession of instants. It is presented in Section 4.

3 Debugging Fair Threads

In this work, we have included debugging support for Fair Threads into BUGLOO, a debugger for Scheme programs compiled into Java VM [12] bytecode. BUGLOO is a complete source-level debugger with a command line language. It is integrated in the Bee development environment [16], and is meant to be used from Emacs or Xemacs.

The tools we have implemented are displayed in a new GUI layer which is used in conjunction with Emacs. It is implemented in Biglook [7].

In this section, we show how to start a debugging session and we present the first two debugging tools we have implemented: an enhanced single stepper and a scheduler and fair thread inspector. They can be used to fix bugs like dead-locks or live-locks that may occur during an instant. In the followings of this paper, the term *debuggee* will denote the program that is being debugged.

3.1 The Fair Threads Debugging Toolbox

A typical debugging session consists in connecting an Emacs buffer with BUGLOO, setting breakpoints somewhere in the source and running the program. Let us suppose that we ran the little program presented in Section 2.3 and that the execution was suspended on a breakpoint line 13. Then, the user can pop up the Fair Threads toolbox showed in Figure 1, from which all the debugging tools are accessible. From top to bottom, the toolbox contains a set of buttons for enhanced single stepping, buttons to display traces of scheduler executions, and a list of fair threads present in the program. We will now describe these tools.

Figure 1. The *Fair Threads* debugging toolbox

3.1.1 Enhanced Single Stepping

Signals and instants introduce new logical points of control in the execution. We have thus enhanced the classic single stepping operation by providing six new possible step points accessible through buttons in the toolbox:

- **End of Instant** continues the execution until the end of the current instant, and suspends the debuggee just before the next instant begins. It is useful to see the state of fair threads or all the broadcast signals at the end of an instant;
- **Beginning of Instant** suspends the execution as soon as a new instant is started. It allows one to quickly step up to a point that will be single stepped more precisely for debug purposes;
- **Thread Switch** suspends the execution as soon as a new fair thread gets the control. It is useful to see how threads are scheduled during an instant;
- **Next Await** continues the execution until any thread awaits a signal. It can be used to single step a communication mechanism;
- **Next Broadcast** continues the execution until a thread broadcasts a signal in the scheduler. It is the dual of the previous step action;
- **Any Scheduler Event** suspends the execution on any of the preceding event.

3.1.2 Trace of Events

Throughout the execution, scheduler events like thread switch, signal await, signal broadcast or end of instant can be recorded. Our interest in tracing these events is twofold:

1. It enhances the debugging information provided by the fair thread inspector that will be presented in Section 3.2. For instance, it allows the debugger to remember which thread is responsible for a particular broadcast, along with its location in the source at this time.
2. It allows one to understand what happened precisely in the scheduler across *many* instants, and to analyze this information *off-line*.

In the debugging toolbox, two options can be checked to control the recording of events.

- **Enhanced inspection**. When checked, the recording of events is activated as soon as the execution is suspended. When the user starts single stepping the program, he automatically gets enhanced information in the inspectors[2]. Enhanced inspection is automatically switched off as soon as the execution is resumed, to avoid performance penalties during normal execution. In this trace mode, recorded events are reset every new instant.
- **Record Scheduler State**. When checked, the event recording stays activated during execution and across instant boundaries. Later, the resulting trace can be cleared or shown by clicking on the appropriate buttons (see Figure 1).

3.1.3 List of Fair Threads

The last part of the debugging toolbox shows the lists of live fair threads (this frame does not show native OS threads present in the program). The list is arranged into a tree where the directory nodes represent the schedulers, and the leaves represent the attached fair threads. Note that there may be several schedulers in a program, and that schedulers can be nested, since they are actually specialized fair threads.

Figure 1 shows the three fair threads present in the previous program, plus their scheduler. Threads are identified by their name, or by a unique thread descriptor that can be used in the BUGLOO command line. Double-clicking on a node opens a *fair thread inspector* in a new window. It is described below.

3.2 Fair Thread Inspector

An inspector provides a graphical representation (henceforth called a *view*) of the state of a debuggee object at the time the execution was suspended. BUGLOO provides various specialized views for different object types. In particular, we have implemented views for the three main types introduced by Fair Threads: schedulers, fair threads and signals. Below, we describe the basic services provided by an object inspector. Then we present the specific Fair Threads views.

3.2.1 Object Inspectors

Inspectors are top-level windows that provide a set of common features and attributes. Figures 2, 3 and 4 show screenshots of different inspectors.

The bottom status bar shows the type of the inspected object. In a view, fields that point to other debuggee objects are themselves inspectable. A common pop-up menu lets the user inspect objects within the current inspector window or in a new one, as show in Figure 2. The toolbar at the top of the view provides a set of generic actions available in every inspector:

- When the user inspects a new object in the same inspector window, the old view is kept in a view history and is accessible through the top toolbar. The history is managed in a browser-like fashion: one can go backward or forward. When a new view is created, it is inserted at the current point in history, and all the views forward this point are forgotten.
- A particular object type can be associated with many graphical views. For instance, a scheduler can be visualized as a simple fair thread or as a scheduler (more specific view). The third button in the toolbox can be used to change the current view (see Figure 3). A default view is provided for all types. It is basically an object inspector and is not presented in this paper.

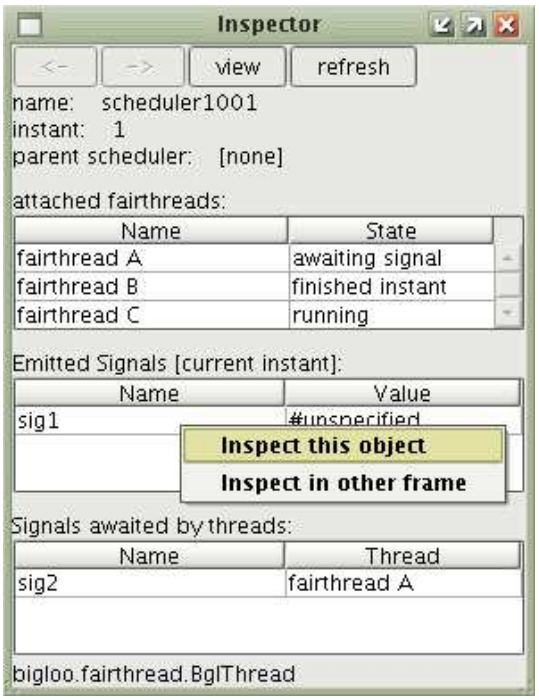


Figure 2. Scheduler Inspector

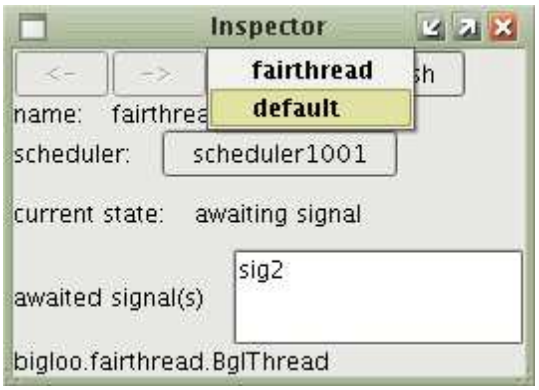


Figure 3. Fair Thread Inspector

3.2.2 The Scheduler View

The screenshot shown in Figure 2 represents the state of the scheduler when the program presented in Section 2.3 is suspended at line 13. The scheduler view is decomposed in four parts. The first part exposes basic information concerning the scheduler: its name, the current instant at the time the execution was suspended, and its par-

[2]Enhanced debugging is only fully effective at the beginning of the next instant.

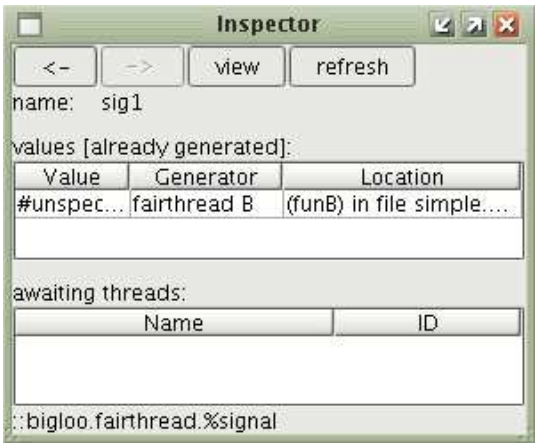


Figure 4. Signal Inspector

ent scheduler, if any (or the symbol `[none]`).

The second part is a table widget that shows all the fair threads attached to this scheduler. Double-clicking on a line pushes a new view of the selected fair thread in the inspector. Information about a fair thread includes its name and its current state in the scheduler. Unlike POSIX threads, a fair thread can be in six different states:

- **running**: the fair thread is currently executing;
- **standby**: the fair thread is eligible for execution during the instant;
- **await**: the fair thread is awaiting signal(s);
- **end of instant**: the fair thread has terminated its execution for the current instant;
- **terminated**: the fair thread has terminated its entire execution;
- **unattached**: the fair thread has not been started yet, because it is not attached to a scheduler. Obviously this state can only be seen in the fair thread view presented further on.

The last two parts of the inspector are devoted to signals.

- A first table represents signals that have been broadcast in the scheduler during the instant. Information about a signal includes its name and its value (or `[...]` if the signal has been broadcast several times). Double-clicking on a line pushes a new view of the selected signal in the inspector.
- A second table represents signals that are awaited by threads, and that have not been broadcast in the scheduler yet. As soon as an awaited signal is broadcast, its entry in the table migrates to the first table. An entry is composed of the signal's name and its awaiting threads.

3.2.3 The Fair Thread View

The fair thread view presented in Figure 3 is quite simple. It first shows the name of the fair thread, and that of its scheduler. If the latter is clicked, a new view is pushed on the inspector. The view also shows the state of the fair thread. It can be any of those presented in the scheduler view. Moreover, if the thread is awaiting one or more signals, their names are displayed in a list-box.

In the screenshot, we see that fair thread `A` is awaiting signal `sig2`. Using many inspectors at a time, one can visualize in detail the state of several fair threads.

3.2.4 The Signal View

The signal view is composed of the name of the inspected signal and of two other tables. The first table contains the different values associated with each broadcast of the signal in the current instant. If enhanced inspection is enabled, each signal broadcast comes with additional information: the fair thread that broadcast the signal and its location in the source at the time of the broadcast. In the screenshot of Figure 4, we see that fair thread `B` has broadcast signal `sig1` from function `funB`.

The second part of the inspector lists the threads waiting for this signal. If the signal has already been broadcast in the instant, the table is empty.

4 Tracing the Scheduling of Fair Threads

We already stated that the Fair Threads framework provides stronger means of synchronization that mutexes, because during an instant broadcast signals are seen by all threads.

In Section 3, we presented a set of tools to address communication or synchronization problems that can occur during a *single* instant. However, the user might need to remember what has occurred several instants backward in time to understand the cause of a particular bug. These tool are not designed to provide such information.

In this section, we present a trace tool that is an effective way to visualize the state of a scheduler *inside* and *between* instants. It gives the user a sharp vision of both the scheduling and the communication between fair threads throughout the execution.

4.1 The Trace Tool

When the trace tool is enabled in the debugging toolbox (Figure 1), the user can display a graphical view of a scheduler execution.

For the sake of the example, let us run the little program presented in Section 2.3 and trace its whole execution. We previously stated that this program never terminates because one thread is waiting for a signal while the others have already terminated.

In presence of a dead-lock, the typical action is to force the suspension of the execution by hitting `CTRL+C`, and then requiring BIGLOO to display the recorded trace. The result is shown in Figure 5. The trace is displayed as a graph:

- The vertical axis shows the fair threads attached to a scheduler and all the signals that were broadcast during the recorded execution slice. Signals always appear at the top, followed by the scheduler[3] and the fair threads.
- The horizontal axis represents the progression of the execution across the instants. Instant boundaries are delimited by thick vertical grey lines, along with their respective number at the bottom.

In the trace view, the execution is decomposed into logical units

[3]The scheduler appears in the trace because it is itself a fair thread.

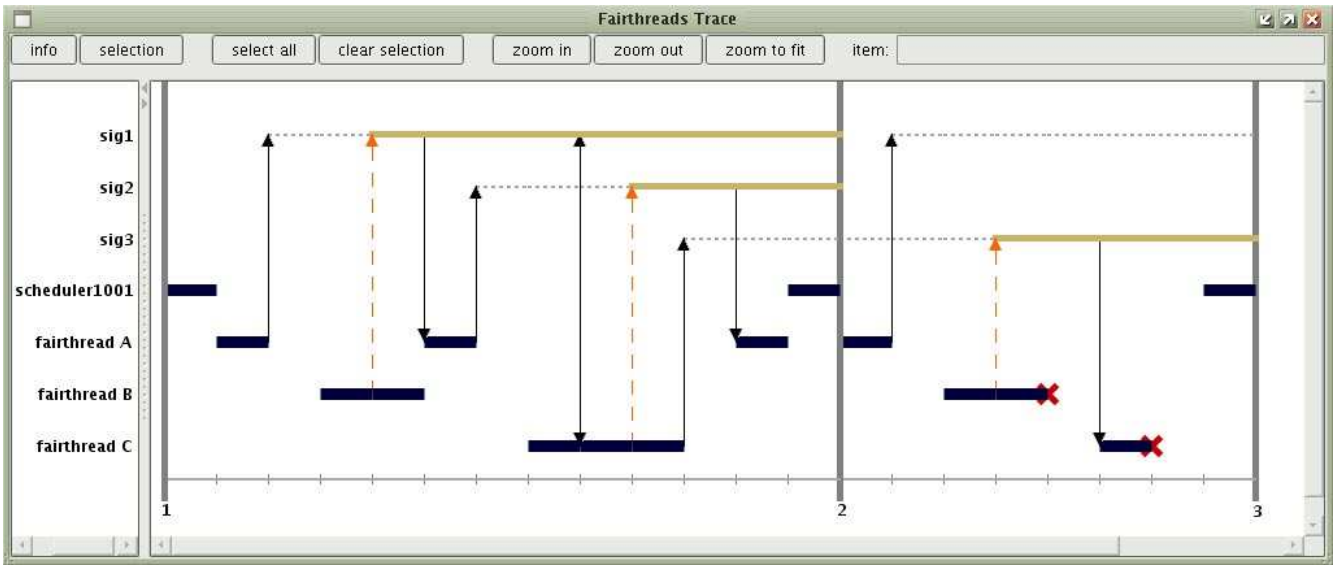


Figure 5. The Trace of the Example shown in Section 2.3

that represent atomic operations that occur inside a scheduler. For example, such units can denote a context switch, the broadcast of a signal, the waiting for a signal, the termination of a thread or the start of an asynchronous I/O operation.

We now explain how to interpret the trace while we describe the important parts of the execution:

At the beginning of instant 1, the scheduler named `scheduler1001` has the control of the execution. The control is symbolized by a thick black horizontal segment. Then, the scheduler allocates the processor to fair thread `A`. This is symbolized by another black segment.

Next, fair thread `A` awaits signal `sig1`, which suspends its execution. The waiting is symbolized by a black vertical arrow that points to the life line of signal `sig1`. A dotted horizontal line is drawn to indicate that this signal has not been broadcast yet during the instant.

Next, the control switches to fair thread `B` which broadcasts signal `sig1` into the scheduler. This is symbolized by a dashed arrow pointing to the life line of `sig1`. To mark the presence of the signal, a thick horizontal line is drawn up to the end of the instant. Remember that broadcasting a signal does not suspend a fair thread. Thus, fair thread `B` has to cooperate explicitly. Overall, it has executed 2 logical operations in the scheduler, hence the double size of the black segment.

Now that signal `sig1` has been broadcast, fair thread `A` is re-elected and continues its execution. The awaking is symbolized by a vertical line starting from the signal life line and pointing to fair thread `A`.

Later in the trace, the control switches to fair thread `C`, which then awaits signal `sig1`. As this signal is already present in the scheduler, the fair thread can continue its execution in sequence. This phenomenon is symbolized in the graph by a double-headed vertical arrow.

When no other fair thread can be scheduled, the scheduler takes back the control and the instant is over.

At the beginning of instant 2, all the signal are reset, thus their respective life lines are empty. Fair thread `A` receives the control and awaits signal `sig1`.

At the bottom right of the graph, one can distinguish two little diagonal crosses at the end of the schedule of fair threads `B` and `C`. This indicates that both threads have terminated their entire execution and will no longer be scheduled.

There is no instant 3. Indeed, the only remaining fair thread in the scheduler is fair thread `A`. Unfortunately, this thread cannot be scheduled because it is awaiting a signal that will never be broadcast from now on. This leads to a dead-lock.

5 Bugloo in Action

In this section we present a complete debugging session on the classical producer-consumer problem. Several producers write data into a global shared buffer of unlimited capacity. Several consumers can read this data and print it using asynchronous I/O operations. We split the complexity of the problem by presenting successively refined implementations, along with the typical synchronization bugs that may arise during this process and how we can track them down with our tools.

5.1 First Implementation

First of all, let us model the problem in Fair Threads. The shared buffer is a simple Scheme list. For the moment, we consider that I/O operations are synchronous. Producers and consumers are naturally modeled as fair threads that put (resp. get) data into (resp. from) the buffer and then cooperate:

```
 1: (define (buffer-fetch)
 2:    (let ((r (car *buffer*)))
 3:       (set! *buffer* (cdr *buffer*))
 4:       r))
 5:
 6: (define (buffer-put! val)
 7:    (if (null? *buffer*)
 8:        (set! *buffer* (list val))
 9:        (set-cdr! (last-pair *buffer*)
10:                  (list val))))
11:

12: (define (make-producer count name)
13:    (make-thread (lambda ()
14:                   (let loop ((n count))
15:                      (put n)
16:                      (thread-yield!)
17:                      (loop (+ 1 n))))
18:                 name))
19:
20: (define (make-consumer name)
21:    (make-thread
22:     (lambda ()
23:        (let loop ()
24:           (print (current-thread) ": " (get))
25:           (thread-yield!)
26:           (loop)))
27:     name))
```

The notification mechanism will occur by the means of the two procedure calls `(put n)` and `(get)`. In the first implementation, the communication model follows a simple wait/notify scheme: on data availability, a signal is broadcast to awake all the consumers.

```
28: (define (wait sig)
29:    (thread-await! sig))
30:
31: (define (notify sig)
32:    (broadcast! sig))
33:
34: (define (put val)
35:    (buffer-put! val)
36:    (notify 'available))
37:
38: (define (get)
39:    (if (buffer-empty?)
40:        (begin
41:           (wait 'available)
42:           (thread-yield!)
43:           (get))
44:        (buffer-fetch)))
```

Note that the `thread-yield!` line 42 is mandatory after the `wait`. Indeed, when a consumer awakes, data may have been already consumed by another consumer. If there was no cooperation, the consumer would retry another `wait` in the same instant. Because a broadcast signal is present until the end of instant, the consumer would not block anymore on signal `available` and would cause a live-lock.

The following piece of program creates producers and consumers and starts the scheduling:

```
(define (start)
   (thread-start! (make-consumer "cons1"))
   (thread-start! (make-consumer "cons2"))
   (thread-start! (make-consumer "cons3"))
   (thread-start! (make-consumer "cons4"))
   (thread-start! (make-producer 1000 "prod1"))
   (thread-start! (make-producer 0 "prod2"))
   (scheduler-start!))
```

When we run the program, the following output is printed on the screen:

```
#<thread:cons4>: 1000
#<thread:cons3>: 0
#<thread:cons2>: 1
#<thread:cons1>: 1001
#<thread:cons2>: 1002
#<thread:cons4>: 2
#<thread:cons3>: 3
#<thread:cons1>: 1003
#<thread:cons2>: 1004
#<thread:cons4>: 4
#<thread:cons3>: 5
```

The first reaction when seeing this output is to think that something went wrong in the scheduling of the producers. Actually, one might assume that producers generated two values in a single instant.

The trace shown in Figure 6 helps to find out why the threads are interleaved this way. It turns out that producers broadcast their signals correctly. In fact, the trace reveals that from an instant to another, fair threads are scheduled in the exact opposite order, which gives the impression of erroneous executions.

In conclusion, we showed that the trace tool is useful to understand the interleaving of threads inside the scheduler. It showed that one should not assume any particular execution order within an instant: the scheduler is deterministic in the sense that another execution will lead to the very same interleaving of fair threads.

5.2 Improving Notification

So far, data availability is signaled to all fair threads. This leads to unnecessary context switches. We can improve the mechanism by putting consumers in a queue, and by signaling availability only to the first thread in this queue. We thus modify the former code as follows:

```
(define (queue-empty?)
   (null? *queue*))

(define (queue-push! val)
   (set! *queue* (append! *queue* (list val))))

(define (queue-pop!)
   (let ((th (car *queue*)))
      (set! *queue* (cdr *queue*))
      th))
```

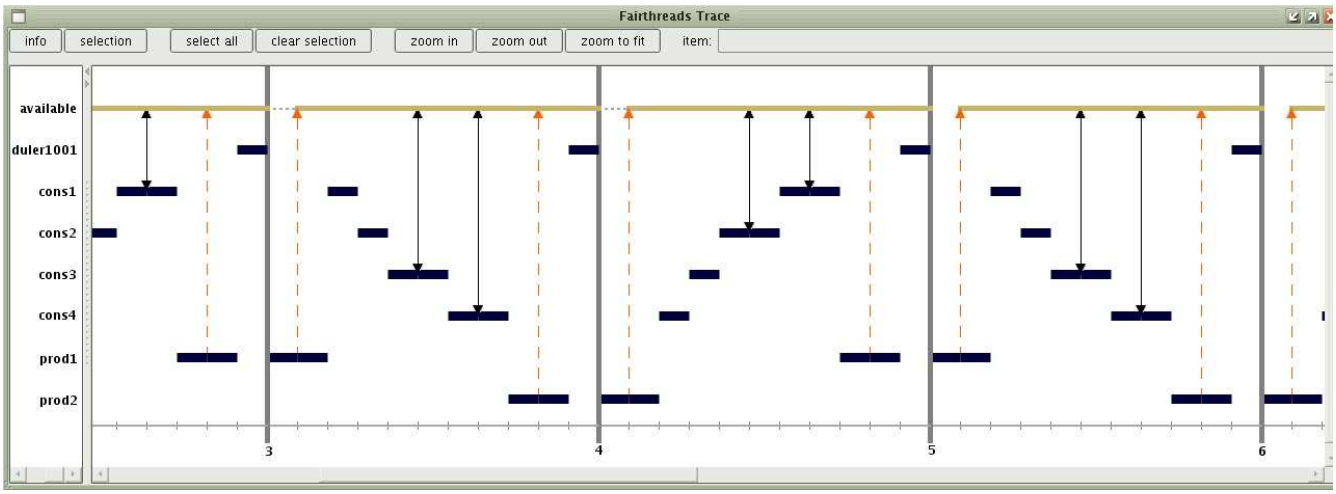


Figure 6. The Trace of the Producers-Consumers with a Naive Implementation.

```
(define (wait sig)
   (let ((self (current-thread)))
      (queue-push! self)
      (thread-await! self)))

(define (notify sig)
   (if (not (queue-empty? sig))
       (broadcast! (queue-pop! sig))))
```

Since any Scheme value can be used to denote signals, we can make each fair thread waiting for a different signal, which is its own thread descriptor returned by `current-thread`. This way, a broadcast signal only awakes one consumer at a time.

5.3 Introducing Non-Determinism

We now replace the consumer's `print` statement at line 24 with a `make-output-signal` to support asynchronous I/O (see Section 2.3.3). This way, the output operation is executed in parallel (*i.e*, preemptively) and does not block other running fair threads, for instance in case the output port is a very slow socket.

We also decide to remove the `thread-yield!` statement line 25, as a call to `make-output-signal` always does an implicit cooperation. The consumer code is rewritten as follows:

```
(define (make-consumer name)
   (make-thread
    (lambda ()
       (let loop ()
          (thread-await!
           (make-output-signal
            (current-output-port)
            (concat (current-thread) ": " (get))))
          (loop)))
    name))
```

When we run the program, the following output is printed on the screen:

```
#<thread:cons4>: 0
#<thread:cons3>: 1000
#<thread:cons3>: 2
#<thread:cons2>: 1002
#<thread:cons4>: 1001
#<thread:cons1>: 1
#<thread:cons1>: 1003
#<thread:cons2>: 3
#<thread:cons2>: 1004
#<thread:cons2>: 5
#<thread:cons3>: 4
```

The output seems coherent. In particular, the new interleaving order and the inversion of data `1002` and `1001` can be explained by the introduction of asynchronous I/O operations. Actually, this modified program contains a subtle bug which will be explained in detail in the following section.

5.4 Profiling the Scheduling

At this time, the program seems bug-free, but the trace tool is still useful to do some profiling analysis. Figure 7 exposes the behavior of the scheduler when running the new program. By observing the trace, we can draw various conclusions:

- The new notification mechanism is working as expected: each producer awakes only one consumer by broadcasting a specific signal.
- Asynchronous I/O operations are materialized in the trace at instant 2 by little diagonal arrows drawn at the extremities of execution segments. An outgoing arrow means that a fair thread started an asynchronous I/O operation and is awaiting its termination. This operation always implies an implicit cooperation. An ingoing arrow means that the consumer can continue its execution because the I/O terminated.
- The trace reveals that asynchronous I/Os can terminate in the instant they were started. This is problematic, because this allowed consumers 1 and 2 to react many times in the same instant and thus to consume more data than they were intended to. We conclude that it was wrong to remove the call to `thread-yield!` line 25 in the consumer code (Section 5.1).
- The trace also reveals that as soon as a consumer is awak-

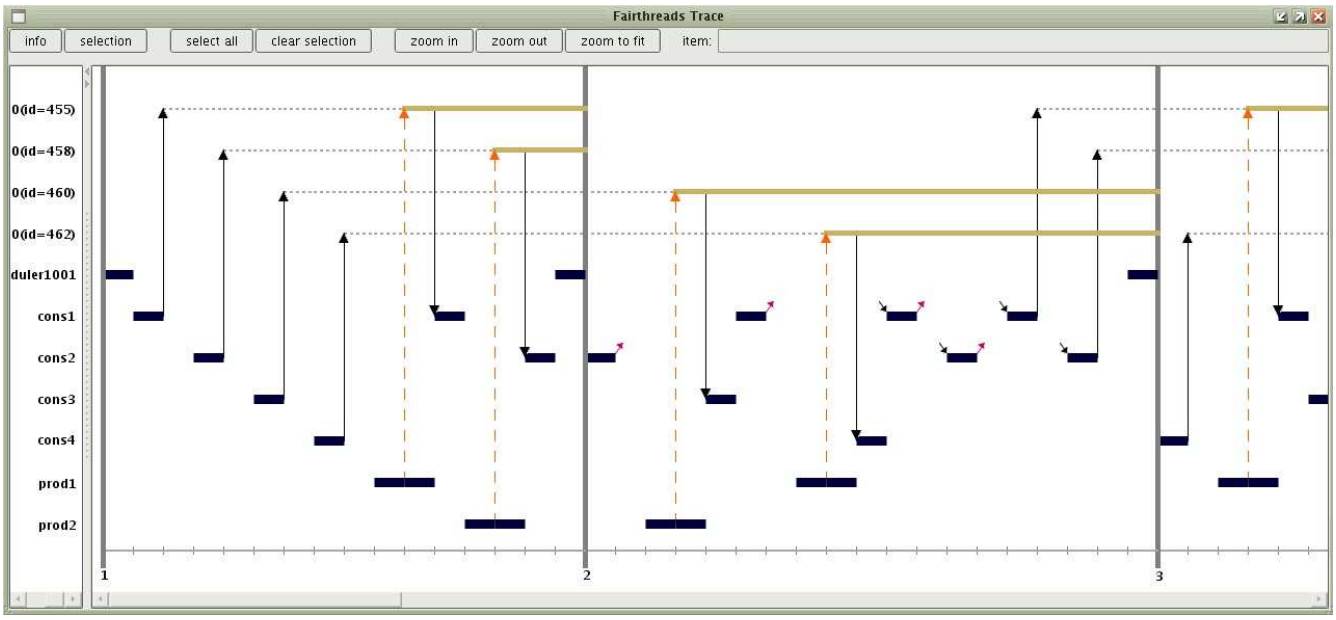


Figure 7. The Trace of the Producers-Consumers with Enhanced Notification and Asynchronous I/Os.

ened, it explicitly cooperates (see `cons2` near the end of instant 1). Then, when it takes back control, it does an asynchronous I/O. The trace tells us that the cooperation is due to the `thread-yield!` located in the `get` function (line 42). In Section 5.1, we stated that this cooperation point was mandatory because a consumer could be awakened while data was no longer available. With the new notification mechanism, this is no longer true. We conclude that this `thread-yield!` leads to superfluous context switches.

In conclusion, we believe that the trace inspector is an effective way to debug communication between threads during execution. It can be used to track down algorithmic bugs, and it can also be employed as a simple profiling tool, since it helps tweaking the cooperation points in a program.

6 Practical Experience

In this section, we briefly explain how the debugger is implemented. We describe the JVM debugging architecture, and how we hook BUGLOO in the debuggee to debug fair threads. In a second part, we present the practical experience we had with our tools and their current limitations.

6.1 Implementation

The debugging architecture of the JVM is close to the one found in GDB [21]: the debugger runs on a JVM, and it instruments the execution of the debuggee which runs in a second JVM. This allows BUGLOO to stay as unintrusive as possible. Debuggee's events like breakpoint hits, single steps or method entries are transmitted to the debugger through an event queue connected to both JVMs. In order to use BUGLOO, it is not necessary to compile the source program in a special debug mode, nor to operate source-code instrumentation on it. The debugger queries remotely the debuggee JVM for information such as a stack trace or the value of a variable. Motivations for such a design have been discussed in detail in a previous paper [2].

To implement fair threads debugging, BUGLOO does not need special hooks in the Bigloo runtime. It only uses some breakpoint trickery and remote stack inspection. For instance, when the user single steps to the end of instant, the debugger sets a temporary breakpoint in scheduler code (somewhere in the Fair Threads runtime), and it tells the debuggee JVM to continue its execution. Later, when the execution breaks into the scheduler, the instant is over and the breakpoint is discarded. The same trickery is employed to implement the trace tool. Additionally, if *enhanced inspection* is required (see Section 3.1.2), the debugger inspects the debuggee's stack frame when such temporary breakpoints are reached, and stores additional information in the trace. An important property of this implementation is that the scheduling of fair threads is never affected by the debugger, since the scheduler is not aware of the instrumentation. In particular, the behavior of the debuggee program is not changed, as opposed to debugging tools that rely on a special interpreter or on source code instrumentation.

6.2 Benefits of the Tools

We have used our tools to debug BUGLOO itself. In the latest version of the debugger, we have modified the GUI so that it does not block anymore while the debugger queries information from the debuggee process. We have to manage a pool of fair threads, and to use techniques such as nesting schedulers. The trace tool helped us to understand that we were waiting a signal in the wrong instant. We also saw that fair threads cooperated too much, leading to superfluous empty instants in the execution.

The Fair Threads API and implementation are subject to change. The scheduler is likely to be re-implemented to avoid unnecessary context switches. For instance, all traces presented in this paper show that the control always returns to the scheduler before the end of instant. Our debugger will certainly help us in making a better implementation.

We believe that the tools we have presented can be very useful for educational purpose. They are simple to use and they help to un-

derstand what is going on in the scheduling of programs. This is a good means to get acclimatized with this style of concurrent programming based on signals and instants.

6.3 Current Limitations

We can detect dead-locks and live-locks, but the latter are currently difficult to deal with. For example, if a fair thread is stuck in an instantaneous loop, repeatedly awaiting a signal that is present in the instant, the trace records a lot of events and may grow too much to be displayed. To prevent this bug, the execution is automatically suspended when an abnormal number of events occurred during a single instant. Nevertheless, the repetitions stay visible in the trace and should be grouped.

Also, while the trace tool is a good way to visualize the communication of threads, it says nothing about the actual processing (like entering functions) done between the communication. This may make the trace difficult to read for programs that do a lot of computation and/or side effects in between synchronizations.

Finally, we have not provided yet a good support for visualizing large programs with many dozens of fair threads running concurrently. We should add means to show or hide signals and threads in the trace. Also, the ability to display long traces in multiple views in a Model-View-Controller fashion would be very useful.

7 Related Work

7.1 Debugging Concurrent Programs in Scheme

Every approach of concurrent programming comes with its specific problems with respect to debugging. Gambit-C 4.0 provides a user space implementation of preemptive threads *à la* POSIX based on continuations. We already discussed the problems inherent to this approach of concurrent programming. PLT's DrScheme [6] provides CML-like [15] concurrent primitives, where threads are meant to execute in independent address spaces with their only communication being via messages sent through channels. However this approach does not solve the problem of direct shared memory access. The thread system found in Scheme 48 [11] is based on optimistic concurrency, which provides a sort of per-thread cached view of the global address space. The use of caches makes it difficult to maintain a valid global state and to visualize it. FrTime [3] implements concurrency with functional reactive programming. It provides signal processors in the spirit of Fran [4] that run in response to "events" such as alarms or messages. To our knowledge, none of the former systems provides a complete tool for debugging concurrency.

In general, tools for debugging concurrent systems suffer from the same difficulty: one has to reason on a program by studying the order in which locks are acquired, or messages are passed. For example, the Concurrent Haskell Debugger [8] allows to visualize graphically the state of CML-like communication channels. The OptimizeIt! JVM profiler can log all the accesses to monitors that occur throughout the execution, to analyze them off-line. On the other hand, when debugging Fair Threads, one can reason on the full *algorithmic* logic of his program (*i.e*, context switches, end of instants, broadcasted/received signals), thanks to sequentiality, determinism and signals. Model checkers [10, 9] are one notable exception: these tools can exhibit complete sequences of execution that lead to a dead-lock or a live-lock. To achieve this, they use techniques such as temporal logics and state space exploration.

7.2 Advanced Traces Visualization

Traces are very effective to debug multi-threaded programs. In GThreads [22], Zhao and Stasko provide a complete set of trace views for graphically depicting the execution of program. One particularly interesting view is the so-called "History View", in which the lifetime of a thread is decomposed into colored segments which represent the functions entered by the thread. Our own trace tool would clearly benefit from this idea.

Jinsight [14, 19] is a Java tool for displaying and analyzing traces of programs. It can generate interactive views that can be unrolled or collapsed. It can also automatically detect patterns in the trace and group them to avoid cycles. We should integrate a similar mechanism into our scheduler traces, to fix the problem of live-lock tracing presented in Section 6.3.

8 Conclusion

In this paper we have presented an extension of the source-level debugger BUGLOO. It provides support for Fair Threads, a new thread-based concurrent programming framework that combines cooperative scheduling and strong communication based on synchronous reactive programming.

We showed that unlike the classic POSIX multi-threading approach, Fair Threads allow to provide the programmer with a strong debugging support. We have described three tools to deal with specific bugs that can arise with Fair Threads. First, an improved single-stepper. Second, a scheduler and signal inspector to analyze the state of threads when the program is suspended. At last, a scheduler tracer to analyze the progression of the scheduling off-line.

The presented tools are new in BUGLOO. We are working on faster ways of recording traces, and on other views that would give more insight on the scheduling activity. In the future, The Fair Threads framework will likely provide means to execute arbitrary computation asynchronously (*i.e*, in preemptive threads). We plan to extend the debugging support for these features.

9 References

[1] F. Boussinot. Java fair threads. Technical Report RR-4139, INRIA, 2001.

[2] D. Ciabrini and M. Serrano. Bugloo: A source level debugger for scheme programs compiled into jvm bytecode. In *Proceedings of the International Lisp Conference 2003*, 2003.

[3] G. Cooper and S. Krishnamurthi. Frtime: Distributed and asynchronous functional reactive programming. Technical Report CS-03-20, Department of Computer Science, Brown University, 2003.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[5] M. Feeley. Scheme request for implementation 18: Multithreading support. http://srfi.schemers.org/srfi-18/srfi-18.html, 2000.

[6] R. B. Findler, J. Clements, M. F. Cormac Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A progamming environment for scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[7] E. Gallesio and M. Serrano. Programming graphical user interfaces with scheme. *Journal of Functional Programming*, 13(5):839–866, September 2003.

[8] C. Grelck and S. Scholz. Axis Control in SaC. In T. Arts and R. Peña, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02)*, volume 2670 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2002.

[9] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.

[10] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[11] R. A. Kelsey and J. A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, 1997.

[13] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. A Nutshell Handbook. O'Reilly & Associates, Inc., 1996.

[14] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12(14):1431–1454, 2000.

[15] J. Reppy. CML: A Higher-order Concurrent Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, number 6 in SIGPLAN Notices, pages 293–305. ACM Press, 1991.

[16] M. Serrano. Bee: an integrated development environment for the scheme programming language. *Journal of Functional Programming*, 10(4):353–395, 2000.

[17] M. Serrano, F. Boussinot, and B. Serpette. Scheme fair threads. In *To appear in the proceedings of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2004.

[18] M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.

[19] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. 2001.

[20] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[21] R. Stallman and R. H. Pesch. *Debugging with GDB: the GNU source-level debugger*. Free Software Foundation, 4.09 for GDB version 4.9 edition, 1993. Previous edition published under title: The GDB manual. August 1993.

[22] Q. A. Zhao and J. T. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95-01, College of Computing, George Institute of Technology, 1995.

Acknowledgments

Many thanks to Bernard Serpette, Frédéric Boussinot, Manuel Serrano, Stéphane Epardaud, Florian Loitsch and to the anonymous reviewers for their helpful feedback on this paper. This document has been typeset in Skribe.

Mobile Reactive Programming in ULM

Stéphane Epardaud
Inria Sophia-Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Stephane.Epardaud@sophia.inria.fr

Abstract

We present the embedding of ULM [7] in Scheme and an implementation of a compiler and virtual machine for it. ULM is a core programming model that allows multi-threaded and distributed programming via strong mobility with a deterministic semantics. We present the multi-threading and distributed primitives of ULM step by step using examples. The introduction of mobility in a Scheme language raises questions about the semantics of variables with respect to migration. We expose the problems and offer two solutions alongside ULM's network references. We also present our implementation of the compiler, virtual machine and the concurrent threading library written in Scheme.

1 Introduction

Today's networks of computers have nothing to do with what we had twenty years ago. While there were very few of them back then, it is now very hard not to be surrounded by more than one computer, practically always connected to some sort of network. And if networks and computers have drastically evolved and multiplied, it is natural that programming languages evolve to exploit their number and interconnections.

The widespread clustering of processors have marked the appearance of parallel multi-threading, while the connectivity phenomenon has brought along distributed programming. Some programming languages nowadays include these features right alongside the `+` and `set!` operations.

However, there are many ways to do multi-threading, and parallel execution is but one of them. Many people accept the common idea of preemptive non-deterministic scheduling and the variety of problems that are bundled along. Deadlocks, race conditions and synchronisation problems are but a few problems that one experiences while taking the perilous learning experience of what we often call *native* threads. Debugging a non-deterministic program is a challenging feat, especially since running it on a single processor does not help.

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

Reactive programming offers the ability to execute multi-threading programs in a concurrent and deterministic way. No more fancy scheduling, surprising race conditions and synchronisation mysteries. Every execution of an otherwise multi-threading program runs according to a precise semantics: the same level of predictability expected from `+` or `set!`.

Distributed programming, much like multi-threading programming has many variants, all of them bearing the unreliability of networks. From communication breakdowns to computer unavailability, the networking part of distributed computing faces non-determinism much in the same way parallel threads do. If distributed programming techniques are to be incorporated into a deterministic language, there has to be a way to isolate any non-determinism in safe and well-known places.

ULM (*Un Langage pour la Mobilité*) is a set of reactive and distributed primitives written by G. Boudol that offers a deterministic semantics for local execution. Following the GALS (Globally Asynchronous Locally Synchronous) model, ULM offers concurrent deterministic multi-threading on each site, together with strong thread mobility while isolating non-determinism.

This paper presents our implementation of a prototype interpreter for a Scheme language augmented with ULM primitives. We introduce the syntax of this new language along with illustrations of how to use the ULM constructs in Scheme. We present the implications of introducing mobility in a Scheme language and how we chose to address them in a way that fits with ULM's objectives. Rather than detail the implementation of our working prototype, we explain the global ideas behind the cooperative scheduling and the mobility of threads.

In Section 2 we present the reactive aspect of ULM. In Section 3 we introduce the mobility primitives, the problems arising from free variables during migration, and the different types of variables we offer as solution. In Section 4 we present an extended example of mobility and multi-threading through agent interactions. The implementation of our prototype compiler and virtual machine is outlined in Section 5. We muse on future directions in Section 6, compare our implementation with related work in Section 7 and finally conclude in Section 8.

2 ULM Reactive Primitives

ULM is inspired by FairThreads [6], which is a reactive variant of Esterel [8], an imperative synchronous language, but you need not

know these languages in order to understand ULM and no prior knowledge will be assumed in this paper.

In reactive programs, execution time is divided in units called *instants*. An instant is a discrete time interval during which threads are allowed to react. The basic idea is that during an instant, all threads that want to run are allowed to run, until they all decide to wait for the next instant (a form of cooperation), or are blocked while waiting for something that is not going to happen during the instant. When all threads are waiting or blocked, we go to the next instant.

2.1 Threads

In order to present the ULM language, we will show some examples of what you can do with it. First of all, we will present the basics of multi-threading in ULM. Its threads are lightweight, cooperative, not parallel but concurrent, do not have their own memory space, and are scheduled in a deterministic fashion. They resemble *event loop* programs, that do not need locks for synchronisation, but need to cooperate at some points to let the other threads run.

Here are two threads that run concurrently (the example is explained below):

```
1: (define (make-printer-thread name)
2:   (ulm:thread
3:     (lambda ()
4:       (let loop ()
5:         (print name)
6:         (ulm:pause)
7:         (loop)))))
8: (make-printer-thread "A")
9: (make-printer-thread "B")
```

In this example we define a procedure (`make-printer-thread`) that creates a new thread that will print its name, cooperate with other threads, and keep doing that forever. We then (lines *8-9*) use that procedure to create two threads: one that will print "A" and the other "B". The result is that "A" and "B" will be printed repeatedly in that order forever.

The procedure *ulm:thread* takes a thunk as parameter, creates a thread that will execute this thunk, and schedules the thread to be run later at the current instant. This program also illustrates that the toplevel execution is in an implicit thread: when the toplevel has finished execution, it implicitly terminates and lets other threads run (in this case, the threads that print "A" and "B"). The concept of cooperation is illustrated in this example with the call to *ulm:pause*, whose effect is to wait for the next instant, explicitly allowing other threads to execute. Calling *ulm:pause* after each thread prints allows the other threads to execute, and makes sure the calling thread won't be awakened until all the other threads are done for this instant, which is when the next instant starts.

2.2 Signals

This example and its following explanation introduce the concept of inter-thread communication:

```
1: (let ((relay (ulm:signal)))
2:   (ulm:thread
3:     (lambda ()
4:       (print "Thread A starts and waits for B")
5:       (ulm:await relay)
6:       (print "Thread A resumes execution")))
7:   (ulm:thread
8:     (lambda ()
9:       (print "Thread B starts and wakes up A")
10:      (ulm:emit relay)
11:      (print "Thread B now terminates"))))
```

In this example, we create a *signal* (*ulm:signal* returns a new signal at line *1*) called `relay` which will serve as a synchronisation and communication means between threads. A signal is a sort of flag that is set to "not there" at the beginning of each instant, and then is set to "present" (or any value, as we will see later) as soon as someone emits it, until the next instant. This allows threads to wait for a signal to be emitted (if it is not present already), and to emit signals to wake up other threads. Here thread `A` starts executing[1], then waits for the `relay` signal (line *5*) to be emitted, which implicitly allows thread `B` to run (since thread `A` is waiting). Thread `B` then runs, and emits `relay` (line *10*), which implicitly allows thread `A` to be awakened and rescheduled later in the instant. Then thread `B` terminates and execution passes to thread `A` (line *6*), which then terminates.

Communication of values between threads is done in a classical manner, through shared variables and synchronisation is ensured via signals. The next version of our ULM interpreter will have an object system in the form of mixins [9], and provide an *Event* mixin that will serve as a signal with an associated value. The next section will discuss the differences in communicating values between threads when migration is involved.

2.3 Suspension

The first of the two main reactive primitives in ULM is *ulm:when*, which introduces suspension. Suspension causes a program to be suspended at each instant when a signal has not been emitted (to put it differently: it allows a program to run only during instants in which a signal is emitted).

We will now illustrate and explain suspension:

```
1: (let ((odd-signal (ulm:signal))
2:       (even-signal (ulm:signal)))
3:   (ulm:thread
4:     (lambda ()
5:       (let loop ((even #t))
6:         (if even
7:           (ulm:emit even-signal)
8:           (ulm:emit odd-signal))
9:         (ulm:pause)
10:        (loop (not even)))))
11:  (ulm:thread
12:    (lambda ()
13:      (ulm:when odd-signal
14:        (lambda ()
15:          (let loop ()
16:            (print "Odd instant")
17:            (ulm:pause)
18:            (loop)))))))
```

This program creates two signals: `odd-signal` and `even-signal`

[1] Because of the deterministic semantics of scheduling, threads are run in their creation order.

that will serve respectively for marking odd and even instants. The first thread loops forever and emits alternatively the `even-signal` or the `odd-signal` at each instant. The second thread enters suspension (line *13*): that is, it is allowed to run only when `odd-signal` is present. At the beginning of each instant it will be blocked until `odd-signal` has been emitted, and only then will it be allowed to run until the next instant. As soon as `odd-signal` has been emitted, the second thread prints something and waits for the next instant (lines *16-17*), which means it will be blocked again until `odd-signal` is emitted again.

2.4 Preemption

The second main reactive primitive is weak preemption, which allows a program to be given up at the end of an instant. Here we introduce preemption and explain it below:

```
 1: (define (run-n-instants n thunk)
 2:   (let ((kill-signal (ulm:signal)))
 3:     (ulm:thread
 4:       (lambda ()
 5:         (let loop ((i n))
 6:           (if (> i 0)
 7:             (begin
 8:               (ulm:pause)
 9:               (loop (- i 1)))
10:             (ulm:emit kill-signal)))))
11:     (ulm:watch kill-signal thunk)))
12:
13: (run-n-instants 3
14:   (lambda ()
15:     (let loop ()
16:       (print "New instant")
17:       (ulm:pause)
18:       (loop))))
```

This example defines a procedure (`run-n-instants`) which takes a `thunk` and a number of instants `n` as arguments. This procedure creates a thread that will wait for `n` instants (lines *6-9*) and then emit a `kill-signal` (line *10*). Before that thread even starts, the procedure will enter a preemption block on `kill-signal` in which it will execute `thunk` (line *11*). The effect of this is that `thunk` will be allowed to run during at most `n` instants because then the killer thread will emit the `kill-signal` which will cause the execution of `thunk` to be aborted at the end of the n^{th} instant.

We then call (line *13*) `run-n-instants` in the implicit thread, with a procedure that prints each new instant. The result is that the implicit thread will print three instants and then return from `run-n-instants`.

There is a minor problem with this code though, because the *ulm:when* and *ulm:watch* blocks terminate immediately upon termination of their body, whether or not the preemption or suspension signal has been emitted. This means that once `thunk` returns (say, a normal return, not interrupted by the `kill-signal`), the killer thread is still running for a number of instants, albeit harmlessly, since emitting the `kill-signal` after `thunk` has returned means that there is nothing to preempt anymore. This is still a waste of execution and a more proper way to implement `run-n-instants` would be such:

```
 1: (define (run-n-instants n thunk)
 2:   (let ((kill-signal (ulm:signal))
 3:         (done (ulm:signal)))
 4:     (ulm:thread
 5:       (lambda ()
 6:         (ulm:watch done
 7:           (lambda ()
 8:             (let loop ((i n))
 9:               (if (> i 0)
10:                 (begin
11:                   (ulm:pause)
12:                   (loop (- i 1)))
13:                 (ulm:emit kill-signal)))))))
14:     (ulm:watch kill-signal thunk)
15:     (ulm:emit done)))
```

In ULM, preemption is said to be *weak* because it does not happen at the moment the preemption signal is emitted during the instant, but only at the end of the instant. This is different from the common notion of preemption using exceptions where no code is executed between the throwing and the raising (or even from *strong* preemption in ESTEREL [8] where the preempted code is not even executed in the first place). In ULM a preempted thread can continue executing at most until the end of the instant. This delaying of preemption (and migration, as we will see later) to the end of instant, as opposed to during the instant, happens because control flow within a given instant should be independent of the scheduling order.

With only the primitives *ulm:thread, ulm:signal, ulm:emit, ulm:when* and *ulm:watch*[2] we can create threads, allow them to cooperate, let them enter critical sections, make them communicate and decide how they should be sequenced.

3 ULM Mobility

ULM defines *strong* mobility primitives that allow *agents* to migrate between *sites*. Strong mobility allows threads to be migrated with their state, without special treatment from the programmer. This type of migration is transparent and the migrated thread doesn't need to notice it migrated. Note the strong and weak variants of preemption are different from the strong or weak notions in mobility. In our implementation, a site is a Virtual Machine (VM), be it on the same computer or on separate networks.

3.1 Mobility

In ULM, an agent is a kind of thread that can move across sites and has an *agent heap*. There are two kinds of heaps in ULM: the classical heap (called *site heap*) is local to a site and does not move. Agent heaps on the other hand are attached to agents, and migrate with them. A variable allocated in either heap is accessed by means of a *reference*, which can be *local* if the reference is on the same site as the heap it points to, or *remote* if they are on different sites. A local reference access is non-blocking while a remote reference access is always blocking. Whenever a thread attempts to access a reference whose heap is remote, it will be blocked until that heap comes in (with its agent) or until the blocked thread is migrated to the heap's site, unless the access is preempted, of course.

Just like a thread is created with (*ulm:thread* `thunk`), we create an agent with (*ulm:agent* `proc`). Migration of an agent is *strong*, and so an agent migrates with its code, registers, stack and heap. Migration is done by calling either (*ulm:migrate-to*

[2]Implementation of *ulm:await* and *ulm:pause* using these primitives is left as an exercise for the reader.

`host)` or (*ulm:migrate-to* `host agent)`. The first form migrates the current agent to the given `host`, while the second form migrates the given `agent` (*subjective* and *objective* migration resp.). While *ulm:thread* takes a thunk for the thread body, *ulm:agent* takes a procedure of one argument for the agent's body. This procedure will be called with the agent's name, which is also returned by the *ulm:agent* call to the creating thread. This name is used for objective migration.

As we mentioned earlier, migration only happens between instants (like preemption), and (*ulm:migrate-to* `host)` does not block until the end of instant. This means that anything executed between the call to *ulm:migrate-to* and the end of instant will be executed *prior* to moving. This is why we often use *ulm:pause* after *ulm:migrate-to*. The reason why subjective migration is non-blocking is to keep it symmetrical with objective migration, which has no reason to be blocking.

Here is a first example of migration:

```
1: (ulm:agent
2:   (lambda (name)
3:     (print "here")
4:     (ulm:migrate-to "other-host")
5:     (ulm:pause)
6:     ;; we are now on 'other-host'
7:     (print "there")))
```

This creates an agent that will print something on its creation site, then migrate (line *4*) and wait for arrival (line *5*). Once it arrives on the new site, it prints something there and terminates.

3.2 Mobility Groups

There is a certain hierarchy between agents and threads: threads have a parent, which can be either the local site, or an agent. Agents on the other hand do not have a parent. Any thread created directly by the implicit thread has the local site as parent, while any thread created directly or indirectly by an agent has that agent as parent. This allows us to form groups of threads that will function and migrate together. Let us illustrate migration grouping:

```
 1: (ulm:thread
 2:   (lambda ()
 3:     (print "Our parent is the local site")
 4:     ; create a new agent and store its name
 5:     (let ((name
 6:            (ulm:agent
 7:              (lambda (name)
 8:                ; create a thread that
 9:                ; stays with us
10:                (ulm:thread
11:                  (lambda ()
12:                    (let loop ()
13:                      (print "Second thread")
14:                      (ulm:pause)
15:                      (loop))))
16:                ; do silly things
17:                (let loop ()
18:                  (print "Agent alive")
19:                  (ulm:pause)
20:                  (loop))))))
21:       ; move the agent and its side-kick
22:       (ulm:migrate-to "host" name))))
```

Here we have a thread created by the implicit thread (line *1*), which has the local site as parent. This thread creates an agent (line *6*) and then migrates it via objective migration (line *22*). This agent will only have its first instant executed on the local site, prior to migration. During that first instant, it creates a second thread (line *10*), and then prints in a loop (lines *17-20*). That second thread will have the agent as parent, and so will migrate with him at the end of the instant. Just like its parent, this thread will only execute locally during its first instant, prior to migration, and so the execution on the local site will be such:

```
Our parent is the local site
Agent alive
Second thread
```

After that, both the agent and its child thread will resume execution and printing on the remote site `host`. Migration groups are used to keep consistency between an agent and the threads it needs to function. It has the additional benefit that these groups keep a local non-blocking access to the agent's references throughout migrations.

3.3 Confinement of Non-Determinism

While the execution of threads and agents within a given instant is entirely deterministic, the physical migration of agents between sites is intrinsically non-deterministic. This is why agents migrate between instants: it isolates non-determinism between instants, at a well defined place. However, sites need not share a global instant. Rather, each site has its local instants, and when an agent changes site, he leaves a local instant to enter another on the new site. Even when sites exchange agents, the number of local instants on each site that lapse during the physical transportation of the agents is arbitrary and non-deterministic. In particular, if an agent A leaves the site S_1 for the site S_2 which is locally at instant I_i (at the time when the agent leaves), the agent can arrive at the instant I_{i+1+n} on S_2, with $n \geq 0$. The inter-instant migration of the agent can be seen as a first local inter-instant phase, and a later second inter-instant phase at the destination site.

3.4 Shared Variables

The behaviour of the first example of migration is quite simple, but migration introduces questions regarding variables that are shared between migrating agents and the threads that stay behind. What happens when two threads sharing a variable are separated is a classical question among mobile languages [2,14]. Let us illustrate one of those problems:

```
 1: (let ((shared-var 2))
 2:   (ulm:agent
 3:     (lambda ()
 4:       (ulm:migrate-to "other-host")
 5:       (ulm:pause)
 6:       ;; we are now on 'other-host'
 7:       (set! shared-var 5)
 8:       (print shared-var)))
 9:   (ulm:pause)
10:   (set! shared-var 9))
```

In this example we create an agent that migrates to `other-host` to set the variable `shared-var` and print it (lines *7* and *8*).

Here we have the local variable `shared-var`, which is free in agent's body: it is allocated in the calling thread's stack. When the agent migrates and keeps using this variable we have a problem: this local variable is being used by two different threads on

different hosts.

Some other languages transform any free variable in an agent's body into a remote *proxy*. Proxies are a way to reference variables across the network in a transparent manner, reading or setting it through the proxy causing network communication between the proxy and the remote variable. The remote references in ULM do not have transparent proxy semantics though. Besides, ULM references are explicitly declared, accessed and read (see Section 3.6), so they are in no way transparent. Furthermore, turning these free variables into references would block the program as soon as line *7*, which is not what one would expect.

Using proxies here would solve this problem, but to what cost? ULM references were created to offer a reliable deterministic semantics of execution locally. This is what GALS means: communications across sites are unreliable and accessing a local variable should in no way introduce non-deterministic behaviour in a thread.

3.5 Migration by Copy

In order to solve the *free variable* problem, we have decided to migrate them by copy. Here is an example to illustrate the copying of free local variables:

```
 1: (define (remote-run remote-host thunk)
 2:   (let ((set-signal (ulm:signal))
 3:         (val 'undef))
 4:     (ulm:agent
 5:       (lambda (name)
 6:         (ulm:migrate-to remote-host)
 7:         (ulm:pause)
 8:         ; we're now on remote-host
 9:         (set! val2 (thunk2))
10:         (ulm:migrate-to "home")
11:         (ulm:pause)
12:         ; we're now back home
13:         (ulm:emit set-signal)))
14:     ; wait for the agent to return
15:     (ulm:await set-signal)
16:     ; return the value
17:     val))
```

This is a first attempt at implementing an RPC (Remote Procedure Call), which unfortunately does not work as intended, as will be explained below. The procedure `remote-run` takes a `remote-host` and a `thunk` as parameters and sends an agent on `remote-host` to execute that `thunk` and return its value. The caller thread waits for the agent to come back by waiting on a shared signal (line *15*), which will be emitted by the agent when it returns (line *13*). Note that executing an agent's body does not yield any value, since it can terminate anywhere and would not know whom to return that value to.

This (wrong) example allocates a `val` variable outside the agent's body (line *3*), which is shared by the agent and the caller thread, but only up to the point when the agent migrates (line *7*). During migration, all free variables used by the agent (underlined in the example) are duplicated for the agent to go along with (marked with $_2$), together with the values associated with those variables at the instant of migration. This is migration by copy. Once the agent comes back home, it still has its own copy of the variable `val` (i.e. $\texttt{val}_2$), which is not the same as the `val` it left behind. Therefore, setting it has no effect for the waiting thread, which will always return an undefined value. We will explain in Section 3.9 why `set-signal` does not suffer from duplication.

With this example, we notice that migration by copy does solve the free variable problem, but is not enough to allow interaction between two threads that have been separated by migration.

3.6 References

This is where ULM references show their value. Let us attempt to solve the last problem with references:

```
 1: (define (remote-run remote-host thunk)
 2:   (let ((ref (ulm:ref))
 3:         (set-signal (ulm:signal)))
 4:     (ulm:agent
 5:       (lambda ()
 6:         (ulm:migrate-to remote-host)
 7:         (ulm:pause)
 8:         ; we're now on remote-host
 9:         (let ((val (thunk)))
10:           (ulm:migrate-to "home")
11:           (ulm:pause)
12:           ; we're now back home
13:           (ulm:ref-set! ref val)
14:           (ulm:emit set-signal))))
15:     ; wait for the agent to return
16:     (ulm:await set-signal)
17:     ; return the value
18:     (ulm:unref ref)))
19:
20: ; go fetch the uptime of "other-site"
21: (remote-run "other-site"
22:   (lambda () (getuptime)))
```

This procedure creates a reference stored on the local site's heap (`(ulm:ref)` creates a new reference, line *2*). It then sends an agent on the `remote-host` (line *6*) to execute the `thunk` there (line *9*) and return (line *10*) with its return value in `val`. Once back, it sets the reference to that value (`(ulm:set-ref! ref var)` affects `val` to the ULM reference `ref`, line *13*), wakes up the caller thread (lines *14* and *16*), which uses that reference to return its value (`(ulm:unref ref)` returns the value of the reference `ref`, line *18*). This example does not illustrate the use of remote references with its blocking semantics, but it does show how references are used by threads separated by migration to share a variable.

In this `remote-run` example, we create a reference to a variable allocated in the local site's heap (line *2*), and the agent migrates with it. During migration, it mutates from a local reference to a remote reference: it becomes a unique distant reference on the other site. But setting it there would block the agent (remember: remote access to references is blocking). Instead, we create a local variable to store the return value of `thunk` (line *9*), and migrate back. Once the agent arrives on the site where the reference is stored, our remote reference becomes a local reference again. Exactly the same reference that was created before leaving, and the same that the caller thread (that stayed here all along) is using. Setting this reference to `val` (line *13*) allows the agent to communicate a value to the caller thread that was waiting for it.

3.7 Global Variables and Modules

Although it may not seem directly relevant to our discussion on mobility, the module system of ULM is presented here because it introduces the definition and scope of global variables. In our implementation of ULM, a global variable is associated to the *module* that declares it. Each module has a list of global variables that can

be *exported* to other modules that *import* them, and a toplevel execution. Here is an example of ULM module:

```
1: (module foo
2:   (import std-scheme ulm)
3:   (export
4:     bar
5:     (gee x)
6:     ))
7:
8: (define bar 2)
9:
10: (define (gee x)
11:   x)
12:
13: (define (mine)
14:   (print "non-exported global"))
```

This declares a module named `foo`, which uses global variables exported by the *ulm* and *std-scheme* modules, and exports the two global variables `bar` and `gee`. Global variables representing closures are exported in a syntax that explicits their prototype (here `(gee x)` to warn importers of `gee` that it is a procedure and takes one argument). Here `mine` is a non-exported global variable containing a closure. It is local to this module and cannot be used by other modules. The exported variables `bar` and `gee` on the other hand can be imported and used by any other module, if they import the module `foo`.

There are two main standard modules in ULM: *std-scheme* which exports some standard Scheme procedures (such as `for-each` or `assq`)[3], and *ulm* which declares and exports every ULM primitive and derived constructs, prefixed with the `ulm:` namespace for clarity.

3.8 Ubiquitous Variables

We now know how to share variables across the network, and migrate with free variables that will be duplicated. What about standard libraries? In the last example, we only talked about free *local* variables, but there are more variables that become free during migration: global variables suffer the same problem. Although you could expect global variables from the current module to be duplicated during migration (just like local variables), variables from other modules (such as our *ulm* or *std-scheme* modules) deserve another treatment: otherwise, migrating any agent would result in migrating copies of whole libraries, which is a waste of bandwidth.

In addition to the local and global variables which are duplicated during migration, there is a type of variable called *ubiquitous*. Ubiquitous variables constitute a category of global variables that are not duplicated during migration. Instead, they are bound dynamically upon arrival on the new site. This imposes a few restrictions though, the first one being that it must be possible to find these variables upon arrival (local variables fall out of this category).

In our implementation of ULM, global variables are bound to modules, and can be exported outside of these modules. Only modules can be declared ubiquitous (using the module declaration `ubiquitous-module` instead of `module` as seen in the last section), which makes all the global variables it exports also ubiquitous. Those modules are called this way because the programmer assumes they can be found everywhere at the same time, that is on every site. Using those ubiquitous variables while migrating means that upon arrival on the new site, they will be dynamically bound to their local counterparts.

Ubiquitous variables allows ULM programs to interact with sites and agents after migration. It is used among other things to move without dragging along whole libraries (such as the standard ones), to be able to call local procedures (like `gethostname`), and to interact with the site or other agents via those local procedures.

Since we declared our *std-scheme* and *ulm* modules ubiquitous, here is what the `remote-run` example looks like with ubiquitous variables underwaved and duplicated variables underlined:

```
1: (define (remote-run remote-host thunk)
2:   (let ((ref (ulm:ref))
3:         (set-signal (ulm:signal)))
4:     (ulm:agent
5:       (lambda ()
6:         (ulm:migrate-to remote-host)
7:         (ulm:pause)
8:         ; we're now on remote-host
9:         (let ((val (thunk2)))
10:           (ulm:migrate-to "home")
11:           (ulm:pause)
12:           ; we're now back home
13:           (ulm:ref-set! ref2 val)
14:           (ulm:emit set-signal2))))
15:     ; wait for the agent to return
16:     (ulm:await set-signal)
17:     ; return the value
18:     (ulm:unref ref)))
19:
20: ; go fetch the uptime of "other-site"
21: (remote-run "other-site"
22:   (lambda () (getuptime)))
23:   )
```

Here it is clear that all the *`ulm:...`* procedures are ubiquitous, as is `getuptime`, since we want to get the uptime of the site to which we migrate. All local variables are duplicated by the migration, as is the global variable `remote-run`, which is not ubiquitous[4].

3.9 Special Values

You will notice that references and signals are underdashed instead of underlined in the previous example. This is because they are in effect duplicated during migration, but their values are special. We already explained that references can change state (local/remote) during migration but remain unique on each site: this is why `ref` is the same as $\texttt{ref}_2$ upon return of the agent (lines *13* and *18* in the last example).

Signal values are also special: they are associated a universal value, which will always be equal after migration, in the same way strings that are not `eq?` can be `equal?`. This explains why emitting $\texttt{set-signal}_2$ (line *14*) awakes the thread waiting on `set-signal` (line *16*).

3.10 How It All Fits Together

We have described informally the behaviour of free local variables, global variables, ubiquitous global variables, and two special kinds

[3] Our ULM Scheme is not R5RS compliant.

[4] Actually, `remote-run` is never used by the agent after migration, so it is not necessary to duplicate it.

of values with respect to migration. It should be noted that aside from the signals and references values, only variables are concerned by migration. In particular, values such as pairs or vectors, which can contain other values, do not act like variables and the distinction between ubiquity or copy is never relevant to values. To illustrate the distinction, two agents cannot share a value through a pair's content unless that pair has been obtained by a common variable since their last migration, and if both agents are on the same site.

Our experience in programming with ULM is that the distinction between these types of variables or values is quite intuitive. Anything you want to keep sharing after migration has to be explicitly declared (through references). The other variables will be taken care of: that is, whether they are dynamic or duplicated, your program will keep running after migration. Suspension on a reference is explicit (via *ulm:ref-set!* for example), so the programmer knows where potential suspension happens.

Whether the variables are dynamic or duplicated is left to the module designer that provides the variable (in the case of global variables), so, for instance the programmer does not need to know (in most cases) whether his implementation of `map` comes from one site or another. In any case, the module designer knows what to declare ubiquitous.

The only thing that might surprise programmers at first is the free variable duplication, if they try to use them as a communication means between migrated agents and sites for example. But this is a habit worth losing in the case of ULM because inter-agent communication can be done via references, signals or dynamic variables. The philosophy behind these types of variables is that local intra-agent execution is the default. Any inter-agent communication is explicitly marked so.

4 Extended Example

We now present an example which illustrates the benefits of ubiquitous variables, along with a mobile reactive chase. The following is a prey/predator example in which rabbits try to escape a fox. Rabbits eat grass in a field (we will use a field per site) until they are fed up or hear a fox arriving or killing another rabbit, in which case they migrate to a random site and leave a trail. The fox makes noise when he arrives in a site, and then hides until rabbits come in or he gives up. When a rabbit comes in the fox kills the first one and goes away. In this example, ubiquitous variables (except those from the *ulm* and *std-scheme* modules) are underwaved:

```
 1: (ubiquitous-module salad-field
 2:   (import std-scheme ulm)
 3:   (export
 4:     kill         ; local signal
 5:     fox-arriving ; local signal
 6:     eat-grass    ; local signal
 7:     (go-away)
 8:     (follow-trail)
 9:     ))
10:
11: (define *trails* '())
12:
13: (define kill (ulm:signal))
14: (define fox-arriving (ulm:signal))
15: (define eat-grass (ulm:signal))
16: (define (go-away)
17:   (let ((dest (random-other-site)))
18:     (set! *trails* (cons dest *trails*))
19:     (ulm:migrate-to dest)
20:     (ulm:pause)))
21:
22: (define (follow-trail)
23:   (let ((dest (if (pair? *trails*)
24:                 (car *trails*)
25:                 (random-other-site))))
26:     (ulm:migrate-to dest)
27:     (ulm:pause)))
```

We define an ubiquitous ULM module with three signals and two procedures exported. These exported variables represent signals and procedures local to a site: in this case a signal to indicate the fox's arrival (`fox-arriving`), one emitted by rabbits while eating (`eat-grass`), and another to represent the fox killing a rabbit (`kill`).

We now define a non-ubiquitous module where the behaviour of rabbits and foxes are each defined in a procedure:

```
 1: (module fox-rabbit
 2:   (import std-scheme ulm salad-field))
 3:
 4: (define (rabbit name)
 5:  (let ((killme (ulm:signal)))
 6:   (let watchout-loop ()
 7:    (ulm:watch killme
 8:      (lambda ()
 9:       (let ((bored #f))
10:        (print name " Rabbit Arriving")
11:        (ulm:watch-or (list fox-arriving kill)
12:          (lambda ()
13:           (let eat-loop ()
14:            (ulm:emit eat-grass killme)
15:            (print name " Rabbit Eating")
16:            (ulm:pause)
17:            (if (= 1 (random 5))
18:             (set! bored #t)
19:             (eat-loop)))))
20:        (if bored
21:         (print name " Rabbit Bored")
22:         (print name " Rabbit Fleeing"))
23:        (go-away)
24:        (watchout-loop)))))
25:   ; we got killed
26:   (print name " Rabbit Dead")))
```

The behaviour of the rabbit agent is defined in the `rabbit` procedure. The rabbit starts by creating a signal (line *5*) by which it will be identified in case it gets killed. It then enters a preemption block on that signal (line *7*): when that signal is emitted, the rabbit dies. In that block it is going to eat (line *13*) for a random amount of instants while watching out for a fox arriving or the fox killing another rabbit (this is a variant of *ulm:watch* which preempts on any presence in a set of signals, line *11*). The eating consists in emitting the `eat-signal` with the signal representing the rabbit's life as value[5] (line *14*). When the rabbit is bored (line *18*) or is preempted by the fox's arrival or killing another rabbit (line *11*), it goes away and loops (lines *23* and *24*).

Notice how `kill, fox-arriving, eat-grass` and `go-away` are ubiquitous variables: they are dynamic per-site.

[5]This introduces valued signals: you can assign several values to a signal during the instant, the first one of which sets it as emitted.

The fox's behaviour is defined in the procedure that follows:

```
 1: (define (fox name)
 2:  (let loop ()
 3:   ; arrive somewhat noisily
 4:   (print name " Fox Arriving")
 5:   (ulm:emit fox-arriving)
 6:   (ulm:pause)
 7:   ; wait for a rabbit silently
 8:   (let wait ((i 20))
 9:    (if (> i 0)
10:     (let ((eaters (ulm:present eat-grass)))
11:      (if eaters
12:       (begin
13:        (print name " Fox Killing "
14:                    (car eaters))
15:        (ulm:emit kill)
16:        (ulm:emit (car eaters)))
17:       (begin
18:        (print name " Fox Hiding")
19:        (wait (- i 1)))))))
20:   (print name " Fox Going")
21:   ; follow the first trail
22:   (follow-trail)
23:   (loop)))
```

The procedure *(ulm:present s)* (line *10*) is used to query the presence of the signal `s` at the current instant. If `s` is emitted during this instant, *ulm:present* will unblock at the current instant and return any value associated with it when it was emitted. But there is no way to know whether a signal will not been emitted within an instant, because we only know its absence when we have decided to stop running threads in an instant. This is called the end of instant, so in ULM, absence can only be determined at the end of instant, and since no thread can run between instants, reaction to absence is always done in the next instant. The behaviour of *ulm:present* when `s` has not been emitted during the current instant is to return `#f` at the next instant.

The fox emits the `fox-arriving` signal in the instant it arrives (line *5*), then it hides and waits for the first broadcast of `eat-grass` (line *10*), which is emitted by any eating rabbit. If there is nothing, that call is blocking (implicit cooperation) and we can safely loop because we're already at the next instant when it returns (line *19*). If there is a rabbit, *ulm:present* returns the list of `killme` signals emitted by each rabbit while eating. Each signal represents the life of a rabbit (the outer `watch` block of his loop, line *7* of the rabbit procedure). The fox can then emit the `kill` signal to warn all rabbits (line *15*), and the signal that will kill the first rabbit that showed up (line *16*). After that, the fox follows the first trail it finds and goes on another site (line *22*).

Note that the killed rabbit is preempted twice in the same instant: once by the emission of its `killme` signal, and the second time by the `kill` signal. Whenever several watch blocks should be preempted, it is the outermost block that is preempted, thus killing the rabbit without making him flee. It is also worth noting that since the preemption is weak, the rabbit still has time to chew his last bit of grass before dying[6].

The use of ubiquitous signals and methods to represent what happens in each site enables us to start rabbits and foxes on any different site, and still have them interact on each site according to the local signals and procedures.

[6]Which clearly shows the total lack of resemblance between these rabbits and Evil ones with Big Sharp Pointy Teeth.

5 Implementation

We chose Scheme as the host language for the ULM primitives in order to concentrate on the reactive and mobility issues and not on complex host language syntax or semantics. Due to the strong migration semantics, we opted for a bytecode compiler/VM couple, to avoid having an interpreter stack while executing ULM programs, since stacks are typically not first-class objects (and indeed not in the target executables our interpreter is compiled in: Java, C, .NET). We use an academic bytecode interpreter from Queinnec [3] to compile scheme primitives to bytecodes and to execute those bytecodes. We also use the syntactic macros from Bigloo [13] along with its implementation of most of the standard Scheme library used from ULM. The compiler and VM are also written in Bigloo Scheme, for portability and native execution.

5.1 Reactivity

Reactive ULM primitives and scheduling are managed in the *ulm* module, implemented in the host language itself, with only three VM primitives added. In the VM, each thread is represented by a closure object, a stack and any register that needs saving by the VM when changing context (such as the program counter, stack pointer, etc...). In the *ulm* module, scheduling is done with an extra thread that is executed only at the end of instants to initiate the next instant. The scheduling of threads during the instant is done by the threads themselves, whenever they emit or wait for signals. In theory the scheduler thread is optional since the *end of instant* could be executed in any other thread, but using an extra thread made things much easier to write and understand.

5.1.1 Contexts

Suspension and preemption are represented by a list of *WW-cells* (When/Watch cells) that are augmented with either a When-cell or a Watch-cell when entering a suspension or preemption block (resp.). When-cells specify the signal of the *ulm:when* block, and a boolean that indicates whether it is satisfied for this instant (it is satisfied if the signal has been emitted). Watch-cells associate the preemption signal with a procedure that can escape from the *ulm:watch* block. Preemption is implemented using `bind-exit`[7]: each time a *ulm:watch* block is entered, we enter a `bind-exit` block and associate its `exit` procedure with the preemption signal.

5.1.2 End of Instant

At the end of the instant, the scheduler thread walks the list of threads and reverts all When-cells to `'unsatisfied`. Then for the outermost Watch-cell that is satisfied, the scheduler notifies the thread that it should be preempted (we will see where this is done later). Any thread with no `'unsatisfied` When-cells (that includes preempted threads) is scheduled to run at the next instant.

5.1.3 Scheduling

We already revealed that intra-instant scheduling is done by the threads themselves, during various ULM primitive calls. Emitting a signal causes the list of threads waiting for it to be examined. Each unsatisfied When-cell is checked and threads that only have satisfied When-cells are rescheduled for later in the instant. In other words, threads that have several When-cells are only allowed to run

[7]A form of `call/cc` whose *escape* procedure is only valid in its dynamic extent.

when all have been satisfied. This enables us to have threads waiting for *n* signals only be present in any one signal queue at a time. A thread will thus hop from one waiting queue to the other (as each queue is satisfied) in the worst case, but be considered for scheduling only once in the best case (if it is in the queue of the its last unsatisfied signal).

5.1.4 Context Switching

Context switching is done when waiting for a signal that hasn't been emitted yet (by entering a *ulm:when* block[8]). When that happens, the thread blocks and finds the next thread to schedule and tells the VM to switch its context to it. Here is the procedure that switches context in the *ulm* module:

```
 1: (define (cooperate)
 2:   ; tell the VM to switch context
 3:   ; to the next thread
 4:   (switch-to-thread (get-next-thread))
 5:   ; treat preemption
 6:   (if (thread-preempted? *current-thread*)
 7:     ; get the Watch-cell of the signal
 8:     ; that preempted us
 9:     (let* ((watch-cell (thread-preempted-cell
10:                         *current-thread*))
11:            (exit (cddr watch-cell)))
12:       (exit))))
```

We can see here that all blocked threads are in the `cooperate` procedure, blocked in the call to the `switch-to-thread` VM primitive. When there is no thread left to schedule, `get-next-thread` returns the scheduler thread, which declares the end of instant.

5.1.5 Preemption

Preemption is decided at the end of the instant, and executed in the next instant. Since all threads unblocked return from the `switch-to-thread` call, we check for preemption there, before returning from `cooperate`. When preemption is needed, we find the `exit` escaper associated with the preempting signal, and execute it, thus unwinding at the end of the `watch` call. This is effectively done in the new instant and ensures that any preemption handlers (such as `unwind-protect`[9]) are called in the new instant and not during the end of instant phase.

5.2 Mobility

Mobility is implemented mostly in the VM, because serialisation of the thread state (stack, memory and bytecodes) requires extensive access to data that should be kept away from the interpreted language. Migrating a thread from one site to another consists in finding all accessed (and future accesses to) variables and bytecodes, modification of bytecodes, serialisation, transport, deserialisation and integration.

5.2.1 Finding Accessible Variables

Finding all accessible variables is done by looking through the current environment, the stack, and the bytecode of all accessible closures. Each accessible object is assigned a unique serial number used later to resolve circular references. Ubiquitous variables are not traversed, since they will be dynamically bound after migration.

5.2.2 Modification of Bytecodes

The bytecode needs to be modified before migration for two reasons: first, the migrating agent will become a special kind of module during migration, and second because this is where variable duplication takes place. Making a module out of each migrating agent allows us to simplify the encapsulating process because an agent migrates with bytecode, a list of constants, a list of global variables and a name, which fits perfectly the role of modules and makes inserting an agent in a site fairly easy. These are special modules however, in the sense that they do not export any variable and cannot be imported. The bytecode modification is needed because during the search for accessible variables, we come across global variables that are not ubiquitous and need special treatment.

There are three types of bytecodes to access variables: `LOCAL-REF/SET`, `GLOBAL-REF` and `IMPORTED-REF`. `LOCAL-REF/SET` represent local variables, that is, lambda parameters, and they can never be ubiquitous. `GLOBAL-REF` and `IMPORTED-REF` are both global variables but the first one is a global variable from the current module and is indexed, while the second one is an imported global variable, referenced by module and global names.

`GLOBAL-REF` bytecodes need to be changed into `IMPORTED-REF` if the current module is ubiquitous, or *phagocytized* otherwise. We call phagocytizing a global variable the action of duplicating it, and adding it to the module we create for the agent migration. In effect the agent keeps using that global variable, but it is relocated in its own module, changes index and migrates with a copy of its current value.

In a similar way, `IMPORTED-REF` bytecodes that refer to non-ubiquitous variables need to be phagocytized and changed into a `GLOBAL-REF`. The closures that are not referenced through ubiquitous variables also need to be phagocytized, since we need to add their bytecode to the agent's module, bytecode which has to be relocated and modified.

5.2.3 Serialisation and Transport

Serialisation is done by iterating all the values we affected serial numbers to, and using either introspection or specialised treatment to create an *alist* structure to represent them. References are serialised specially, by mutating their state to remote where applicable: all local references not allocated in the agent's heap become remote before migrating. Local references that stay behind but point to the migrating agent's heap also become remote. These serialised values, along with the agent's module and a pointer to the agent's thread structure (which happens to be the root of serialisation) are then sent along the network asynchronously.

5.2.4 Deserialisation

On the other site, an asynchronous thread waits for incoming agents and stores them during the ULM instants until synchronous incorporation at the end of instant phase. Deserialisation is done in two phases: allocation of all the objects that have a serial number, and affectation of all these object's members that were referred to by serial number. Reference mutation also happens during this phase: remote references held by the agent that point to a local heap be-

[8] *ulm:pause* and *ulm:await* are derived from *ulm:when*.

[9] A variant of `dynamic-wind` with no *before* block.

come local, as do remote references present on the site that point to the new agent's heap.

5.2.5 Integration

Integration is the simplest phase, since after deserialisation we are left with a pointer to the agent thread, and its module. We simply load that module, add the agent thread to the list of threads and notify the *ulm* library that there is a new agent to schedule.

5.3 Migration Examples

Now that we have seen all the details of global variables and migration, here is an example of how migration actually works, as far as the programmer is concerned:

```
 1: (module home-mod
 2:  (exports
 3:   home-var
 4:   (home-fun arg)))
 5:
 6: (define home-var 3)
 7:
 8: (define (home-fun arg)
 9:   (set! home-var arg))

10: (ubiquitous-module ubiq-mod
11:  (import home-mod ulm)
12:  (exports
13:   ubiq-var
14:   (ubiq-fun a b)))
15:
16: (define ubiq-var "dynamic")
17:
18: (define (ubiq-fun a b)
19:   (* a b))
20:
21: (ulm:agent
22:  (lambda ()
23:    (ulm:migrate-to "host")
24:    (ulm:pause)
25:    (print ubiq-var)
26:    (ubiq-fun home-var 2)))

27: (module main-mod
28:  (import home-mod ubiq-mod ulm))
29:
30: (define my-var "hello")
31:
32: (ulm:agent
33:   (lambda ()
34:     (home-fun 5)
35:     (ulm:migrate-to "host")
36:     (ulm:pause)
37:     (print my-var)
38:     (ubiq-fun 2 3)))
```

We define three modules: `home-mod` exports non-ubiquitous global variables, `ubiq-mod` defines and exports ubiquitous global variables and sends an agent to `host`, while `main-mod` defines non-ubiquitous variables and also sends an agent to `host`. After the transformations of bytecode during migration, here is how the two agents would look like upon arrival on *host* if their bytecode was disassembled into this fictitious code:

```
 1: (agent-module agent1
 2:  (import ubiq-mod ulm))
 3:
 4: (define home-var 5)
 5:
 6: (define __agent1-body
 7:  (lambda ()
 8:    (ulm:migrate-to "host")
 9:    (ulm:pause)
10:    (print ubiq-var)
11:    (ubiq-fun home-var 2)))

12: (agent-module agent2
13:  (import ubiq-mod ulm))
14:
15: (define home-var 5)
16:
17: (define (home-fun arg)
18:   (set! home-var arg))
19:
20: (define my-var "hello")
21:
22: (define __agent2-body
23:   (lambda ()
24:     (home-fun 5)
25:     (ulm:migrate-to "host")
26:     (ulm:pause)
27:     (print my-var)
28:     (ubiq-fun 2 3)))
```

This example illustrates several things. First, that the agents become their own module (illustrated by the fictitious `agent-module` directive). To their modules are added copies of any non-ubiquitous global variable they were using (from their module or any other): what we call phagocytizing. All ubiquitous global variables (like `ulm:pause` or `ubiq-fun`) mutate (if not already) into a dynamic variable bound upon arrival to the new site's equivalent variables.

Note that while agent modules are a practical implementation technique, they are not directly available to the programmer, who will likely never need to know about them.

6 Food For Thought

The ULM interpreter we have implemented is a prototype: it implements the semantics of Scheme and the ULM primitives, and adds the notion of ubiquitous variables for the migration semantics. However, there are a number of things that still need to be worked on: better compilation analysis and optimisation and a distributed garbage collector. The possibility to call native code from ULM and have that native code call back ULM code has been implemented, and we are studying its implications regarding mobility.

Some other enhancements would benefit directly to the ULM primitives: during compilation, bytecodes for global variable access could be adapted for ubiquitous modules (this is currently done during migration), as long as it does not impede on non-mobile execution (since we expect migration to be less frequent than global variable access). Better code analysis and compilation could lead to reduced memory traversal during migration (unused variables or dead code for example).

Implementation of derived ULM procedures (such as `ulm:pause` or `ulm:present`) would benefit from direct support in the reactive engine, instead of being implemented via ULM primitives.

An object system in ULM could allow us to implement classical distributed proxies that could be used in migration to implement dif-

ferent argument passing styles for RPC (copy, migrate, visit, lazy, etc...). Representing signals as objects could also open new ways to interact with them. A preliminary work on implementing a mixin object system as defined by G. Boudol [9] has been done, but is not presented in this paper.

Mechanisms in case of migration transport failure or node unavailability have not been studied yet, as these types of failures are hard to represent semantically. We could imagine that migration would return a signal that would be emitted in case of successful arrival, with a notion of timeouts and retries (any agent that can migrate can also be saved on disk[10] to be used for retries or reentry).

The notion of ubiquitous modules implies that an agent expects them to be on every site it visits. Mechanisms in case of missing ubiquitous module upon arrival have not been studied. Automatic retrieval, migration failure or blocking the agent are possible answers, although the last one fits best the philosophy behind ULM. Reifying modules as first-class objects could also provide clues on how to treat this, as agents could perhaps fetch and load modules explicitly during execution.

Although the ULM primitives integrate well with `bind-exit` and `unwind-protect` (both presented in Section 5.1.1), this is because `bind-exit` can be seen as an intra-instant preemption akin to *`ulm:watch`*. `call/cc` on the other hand, interferes with ULM primitives, and introduces questions regarding the passing of continuations between threads which we do not wish to allow.

Even though an agent can acquire new procedure values during migration, which means the agent will collect the bytecode, together with the captured and global variables of that procedure, each site's GC ensures that they will be collected when not used anymore. In any case, the traversal done prior to migration will not collect unused procedures, and dead code analysis will help in leaving behind any variable access that will never be reached in the bytecode that needs to be taken. There is no reason why agents would grow upon each migration unless they need to by collecting useful procedures. The captured bytecode could however often be shared between agents and even with local sites, but we have not studied such a mechanism for our prototype.

7 Related Work

Few other languages offer both Mobility and Reactivity together with a strong deterministic semantics. Junior [6] offers both a deterministic semantics and reactive programming but only reactive mobility: the state of the reactive engine can be migrated, but not the state of the host language (Java in this case). In Junior, this does not have a big impact, since aside from the reactive instructions, non-reactive Java code is supposed to be atomic and have finished execution between instants. ULM on the other hand allows reactive instructions (including migration) to be called by non-reactive instructions.

ESTEREL [8] is the reactive language from which most reactive primitives in ULM are inspired. Its model of execution is however very different from ULM because it is based on calculation rather than discovery. In our model we discover emitted signals as we execute the code that emits them, while in ESTEREL the presence or absence of signals is calculated for each instant, and holds throughout the entire instant. Because of that, ESTEREL is less modular and dynamic than ULM. It also does not provide mobility.

[10]This is the notion of *checkpoints* in Eden [1] and Emerald [5].

Bigloo FairThreads [6] add a reactive library to Bigloo Scheme (on which parts of our reactive module are based), but do not support migration. On the other hand, FairThreads support multiple schedulers and asynchronous threads (that interact in a deterministic way with synchronous threads by becoming synchronous during interaction).

Concurrent ML [11] offers a preemptive scheduling of multiple threads in a functional language. Communication and synchronisation between threads in CML is done via channels that can be shared by multiple threads, whereas ULM synchronisation is done via broadcast events. It should be possible to program in a similar way to ULM's signals, but critical sections have to be explicitly marked, as with traditional preemptive schedulings. CML does not seem to have a preemption mechanism and does not offer mobility in the language.

Obliq [12] is a functional language that supports strong mobility and remote references through *proxies*. The semantics of their migration is to transform every free variable (here, all variables defined prior to migration become free) into remote references. Obliq also supports threads but the parallel, non-deterministic kind.

Kali Scheme [10] proposes mobile procedures that serve as RPC, with the client providing the server with the procedure it wants it to run. The remote procedure shares memory with the caller via *Address Spaces* and proxies, which is comparable to ULM's heaps and references, except that ULM's remote access is blocking while Kali Scheme is proxied. Thread mobility is done by capturing the current continuation of the thread and executing it remotely. Their migration of the stack reuses a concept found in Emerald [5] where only the top stack frames are migrated, while the rest are migrated on-demand.

Erlang [4] is a functional language aimed at distributed programming. It provides mobility in the same form as Kali Scheme, by spawning an asynchronous process in a remote host by sending a procedure there. Communication and synchronisation between threads (remote or local) is done through messages and queues with unidirectional and synchronous communication. The scheduling of Erlang threads is preemptive, so critical sections have to be explicitly marked, and the execution of non-perfect critical sections is non-deterministic and suffers from the usual debugging problems.

8 Conclusion

In this paper we have presented the ULM primitives and how to use them to implement multi-threading and mobile programs that interact with other unknown programs. We have shown the questions mobility poses regarding variable access, and have proposed different solutions to address them while preserving the local execution reliability that ULM offers. We introduce duplicated and dynamic variables for the Scheme implantation of the ULM primitives we implemented, and show how this is intuitive for the programmer. At the same time, we show how to use dynamic variables to interact with other unknown agents.

Our prototype implementation of the compiler and virtual machine serves as proof-of-concept for the ULM specification, and enables us to implement functionalities as complex as RPCs with very few lines of code. Indeed, we believe the set of primitives ULM provides is powerful enough to implement different types of distributed programming techniques, while providing a clear and predictable framework.

9 Bibliography

[1] A.P. Black – **The Eden Programming Language** – Technical Report 85-09-01, Dept. of Computer Science, University of Washington, Seattle, Washington, September, 1985.

[2] Alfonso Fuggetta and Gian Pietro Picco and Giovanni Vigna – **Understanding Code Mobility** – IEEE Trans. Softw. Eng., 24, (5), 1998, pp. 342–361.

[3] Christian Queinnec – **Lisp in Small Pieces** – *Cambridge University Press*, 1996.

[4] ERLANG – **http://www.erlang.org**.

[5] Eric Jul and Henry Levy and Norman Hutchinson and Andrew Black – **Fine-Grained Mobility in the Emerald System** – ACM Transactions on Computer Systems, 6, (1), New York, NY, USA, February, 1988, pp. 109–133.

[6] FairThreads – **http://www-sop.inria.fr/mimosa/rp/FairThreads** – *MIMOSA - INRIA*.

[7] G. Boudol – **ULM: a core programming model for global computing** – Proceedings of ESOP 04, Lecture Notes in Computer Science (LNCS), 2004, pp. 234-248.

[8] Gerard Berry – **Constructive Semantics of Esterel: From Theory to Practice (Abstract)** – Algebraic Methodology and Software Technology, 1996, pp. 225.

[9] Gérard Boudol – **The Recursive Record Semantics of Objects Revisited** – Proceedings of the 10th European Symposium on Programming Languages and Systems, 2001, pp. 269–283.

[10] Henry Cejtin and Suresh Jagannathan and Richard Kelsey – **Higher-Order Distributed Objects** – ACM Transactions on Programming Languages and Systems, 17, (5), September, 1995, pp. 704–739.

[11] John H. Reppy – **Concurrent Programming in ML** – *Cambridge Univ Press*, 1999.

[12] Luca Cardelli – **A language with distributed scope** – Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1995, pp. 286–297.

[13] Manuel Serrano and Pierre Weis – **Bigloo: A Portable and Optimizing Compiler for Strict Functional Languages** – Static Analysis Symposium, 1995, pp. 366-381.

[14] Tatsurou Sekiguchi and Akinori Yonezawa – **A calculus with code mobility** – Proceeding of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems, 1997, pp. 21–36.

Shift to control

Chung-chieh Shan
Harvard University
ccshan@post.harvard.edu

Abstract

Delimited control operators abound, but their relationships are ill-understood, and it remains unclear which (if any) to consider canonical. Although all delimited control operators ever proposed can be implemented using undelimited continuations and mutable state, Gasbichler and Sperber [28] showed that an implementation that does not rely on undelimited continuations can be much more efficient. Unfortunately, they only implemented Felleisen's `control` and `prompt` [18, 19, 21, 22, 49] and (from there) Danvy and Filinski's `shift` and `reset` [11–13], not other proposed operators with which an expression may capture its context beyond an arbitrary number of dynamically enclosing delimiters.

We show that `shift` and `reset` can macro-express `control` and `prompt`, as well as the other operators, without capturing undelimited continuations or keeping mutable state. This translation is previously unknown in the literature. As a consequence, research on implementing `shift` and `reset`, such as Gasbichler and Sperber's, transfers to the other operators. Moreover, we treat all these operators by extending a standard CPS transform (defying some skepticism in the literature whether such a treatment exists), so they can be incorporated into CPS-based language implementations.

1 Introduction

The *continuation* is the rest of the computation, represented by the *context* of the current expression being evaluated. For example, in the program

```
(cons 'a (cons 'b (cons 'c '())))
```

the continuation of `(cons 'c '())` is to `cons` the symbol `b`, then the symbol `a`, onto the intermediate result. This continuation is represented by the context `(cons 'a (cons 'b _))`, where _ is a hole waiting to be plugged in.

Continuations can exist in a program at two levels. First, code may be written in *continuation-passing style* (CPS), in which continuations are managed explicitly as values at all times. Second, the underlying control flow of a program can be treated in terms of continuations. Scheme provides `call-with-current-continuation` (hereafter `call/cc`) to access these implicit continuations as first-class values [35]. Implicit continuations can be made explicit by a CPS transform on programs; explicit continuations can be made implicit by a corresponding *direct-style* transform [7, 14, 15, 46].

A *delimited* (or *composable*, or *partial*) continuation is a prefix of the rest of the computation, represented by a delimited part of the context of the current expression being evaluated. For example, in the program

```
(cons 'a [(cons 'b (cons 'c '()))])
```

the continuation of `(cons 'c '())`, as delimited by the square brackets, is to `cons` the symbol `b` onto the intermediate result. This delimited continuation is represented by the delimited context `[(cons 'b _)]`.

Delimited continuations, like undelimited ones, can be explicit (in CPS code) or implicit (in direct-style code). Since Felleisen's work [18, 19], many control operators have been proposed to access implicit delimited continuations as first-class values. A typical proposal provides, first, some way to delimit contexts, and second, some way to capture the current context up to an enclosing delimiter. For example, Danvy and Filinski [11–13] proposed two control operators `shift` and `reset`, with the following syntax.

$$\text{Expressions} \quad E ::= \cdots \mid \texttt{(shift f } E\texttt{)} \mid \texttt{(reset } E\texttt{)} \qquad (1)$$

Contexts are captured by `shift` and delimited by `reset`. More specifically, `shift` captures the current context up to the nearest dynamically enclosing `reset`, replaces it *abortively* with the empty delimited context [_], and binds `f` to the captured delimited context as a functional value. For example, the program

```
(cons 'a (reset (cons 'b
  (shift f (cons 1 (f (f (cons 'c '()))))))))
```

evaluates to the list `(a 1 b b c)`, because `shift` binds `f` to the value `(lambda (x) (reset (cons 'b x)))`, which represents the delimited context `[(cons 'b _)]` captured by `shift`. At the same time, `shift` also removes that context from evaluation—in other words, it *aborts* the current computation up to the delimiting `reset`—so the result is not `(a b 1 b b c)`.

Continuations have found a wide variety of applications. Delimited continuations, in particular, have been used in direct-style representations of monads [23–25], partial evaluation [8, 17, 26, 38, 52], Web interactions [29, 43, 44], mobile code [50], the CPS transform

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

itself [11–13], and linguistics [3, 47]. However, the proliferation of delimited control operators remains a source of confusion for users and work for implementors. Even though all delimited control operators in the literature can be implemented using `call/cc` and mutable state, we would prefer a *direct* implementation—that is, an implementation that does not rely on undelimited continuations—in hope of reaping the efficiency gains recently shown by Gasbichler and Sperber [28] with their direct implementation. Unfortunately, Gasbichler and Sperber only implement Felleisen's `control` and `prompt` [18, 19] and (from there) Danvy and Filinski's `shift` and `reset` [11–13], not other proposed operators that allow an expression to capture its context beyond an arbitrary number of dynamically enclosing delimiters [30–32, 45]. Although it is clear that the latter operators can macro-express[1] the former ones in pure Scheme without `call/cc` or `set!`, the converse "seems not to be known" [30, 31]. Hence it is unclear how an improved implementation of `shift` and `reset`, such as Gasbichler and Sperber's, can help us implement other control operators better.

Because the "static" control operators `shift` and `reset` correspond closely to a standard CPS transform [12], to macro-express other, "dynamic" control operators in terms of `shift` and `reset` is to extend that transform. In the literature, dynamic control operators like `control` and `prompt` are often treated, as if by necessity, using a non-standard CPS transform in which continuations are represented as sequences of activation frames [21, 22, 42]. By contrast, we show in this paper that a standard CPS transform suffices, as one might expect from Filinski's representation of monads in terms of `shift` and `reset` [23–25] (see Section 3.1). What distinguishes dynamic control operators is that the continuation is *recursive*. Thus, in a language supporting recursion like (pure) Scheme, `shift` and `reset` can macro-express the other control operators after all. As a consequence, any direct implementation of `shift` and `reset`, such as Gasbichler and Sperber's, gives rise to a direct implementation of the other operators.[2] Moreover, because our translation of all these operators extends a standard CPS transform, they can be incorporated into CPS-based language implementations.

[1] By "macro-express" we mean Felleisen's notion of macro expressibility [20], but we surround each program by a "`top-level`" construct to mark its syntactic top level. We also impose an additional requirement: given any space consumption bound s, there must exist another space consumption bound s', such that every program within s translates to a program within s'. This requirement is intended to rule out

- implementing delimited continuations by capturing undelimited ones; and
- keeping mutable state by modeling memory in a single storage cell, which `shift` and `reset` can simulate (while accumulating garbage in the simulated store).

Space consumption can be defined along the lines of Clinger [5], for an abstract machine such as Biernacka et al.'s for `shift` and `reset` [4].

[2] A reviewer suggests that Gasbichler and Sperber's technique can be easily adapted to other control operators. For example, to implement the (dynamic) `shift0` operator below, it seems that one need only replace the reset flag in every frame with a reset count, and decrement it after shifting. Given how many delimited control operators have been (and will be?) proposed—several, like `cupto` [30, 31], are related but not identical to the four considered in this paper—macro-expressibility results like ours are attractive because they do not require changing the Scheme implementation at all before new operators can be introduced.

The rest of this paper is structured as follows. Section 2 introduces the static control operators `shift` and `reset`, and their dynamic counterparts. Section 3 expresses dynamic control in terms of static control with recursive continuations. Section 4 then concludes and mentions additional related work.

2 A tale of two resets

Danvy and Filinski's `shift` and `reset` [11–13] can be defined operationally as well as denotationally. Operationally, we can specify transition rules in the style of Felleisen [18]:[3]

$$M[\texttt{(reset } V\texttt{)}] \rhd M[V] \tag{2}$$

$$M[\texttt{(reset } C[\texttt{(shift f } E\texttt{)}]\texttt{)}] \rhd M[\texttt{(reset } E'\texttt{)}] \quad \text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (x) (reset } C[\texttt{x}]\texttt{))}\} \tag{3}$$

Here V stands for a value, C stands for an evaluation context that does not cross a `reset` boundary, and M stands for an evaluation context that may cross a `reset` boundary:

$$\text{Values} \qquad V ::= \texttt{(lambda (x) } E\texttt{)} \mid \cdots \tag{4}$$

$$\text{Contexts} \qquad C[\,] ::= [\,] \mid C[\texttt{(}[\,]\ E\texttt{)}] \mid C[\texttt{(}V\ [\,]\texttt{)}] \mid \cdots \tag{5}$$

$$\text{Metacontexts} \qquad M[\,] ::= C[\,] \mid M[\texttt{(reset } C[\,]\texttt{)}] \tag{6}$$

Denotationally, we can specify a CPS transform to map programs that use `shift` and `reset` to programs that do not. The core of this transform is shown in Figure 1; its first three lines are what this paper means by "a standard (call-by-value) CPS transform".[4]

As Danvy and others have long observed [10], the syntactic definitions above of contexts and metacontexts are not rabbits out of hats. Rather, contexts are defunctionalized representations of the continuation functions in Figure 1.

Contexts of the form:	represent continuations of the form:
$[\,]$	`(lambda (v) v)`
$C[\texttt{(}[\,]\ E\texttt{)}]$	`(lambda (f)` $(E'$ `(lambda (x) ((f x)` $C'))))$
$C[\texttt{(}V\ [\,]\texttt{)}]$	`(lambda (x) ((`V' `x)` $C'))$

Similarly, metacontexts (such as `(reset (f (reset (g [ ]))))`) are defunctionalized representations of the implicit metacontinuations in Figure 1—that is, of the continuations that can be made explicit by CPS-transforming the right hand side of Figure 1.

The CPS transform relates not just terms but also types between the source and target languages. If the source program is a well-typed term in, say, the simply-typed λ-calculus, then the output of the transform is also well-typed in the simply-typed λ-calculus: every source type at the top level or to the right of a function arrow is

[3] To help the exposition below, these transition rules do not handle the case when a `shift` term is evaluated with no dynamically enclosing `reset`. Danvy and Filinski's original proposal amounts here to enclosing the entire program in a top-level `reset`.

[4] The right-hand-sides for `shift` and `reset` in Figure 1 contain non-tail calls, as do (18–19) in Section 3.1 below. Thus these equations do not really constitute a CPS transform, only a continuation-*composing*-style transform that extends a standard CPS transform on the pure λ-calculus. In particular, the output of this transform is sensitive to the evaluation order of the target language. Danvy and Filinski [12] regain CPS by CPS-transforming the output of this transform a second time. We can do so but need not, since by Section 3.2 our equations' right-hand-sides will be in CPS again, with all arguments pure.

$$\overline{\texttt{x}} = \texttt{(lambda (c) (c x))}$$
$$\overline{\texttt{(lambda (x)}\ E\texttt{)}} = \texttt{(lambda (c) (c (lambda (x)}\ \overline{E}\texttt{)))}$$
$$\overline{(E_1\ E_2)} = \texttt{(lambda (c) (}\overline{E_1}\ \texttt{(lambda (f) (}\overline{E_2}\ \texttt{(lambda (x) ((f x) c))))))}$$
$$\overline{\texttt{(reset}\ E\texttt{)}} = \texttt{(lambda (c) (c (}\overline{E}\ \texttt{(lambda (v) v))))}$$
$$\overline{\texttt{(shift f}\ E\texttt{)}} = \texttt{(lambda (c) (let ((f (lambda (x) (lambda (c2) (c2 (c x))))))}\ (\overline{E}\ \texttt{(lambda (v) v))))}$$

Figure 1. A continuation-passing-style transform for `shift` and `reset`

mapped to a type of the form $(\tau \to \omega_1) \to \omega_2$, where ω_1 and ω_2 are *answer types* [39]. Moreover, the type system of the target language can be regarded as a type system for the source language. For example, the expression

```
(shift f (if (f 'a) 1 2))
```

translates to a term of the type $(\mathsf{Sym} \to \mathsf{Bool}) \to \mathsf{Int}$. In words, the expression can appear in a context that produces a boolean when plugged with a symbol, and produce an integer as the final answer. We can take such descriptions as the types of source terms, as Danvy and Filinski [11] do. They write the typing judgment

$$\cdot, \mathsf{Bool} \vdash \texttt{(shift f (if (f 'a) 1 2))} : \mathsf{Sym}, \mathsf{Int} \qquad (7)$$

to mean that the expression behaves locally like a symbol, but incurs a control effect that changes the answer type from Bool to Int.

The transition rule (3) for `shift` mentions `reset` twice on its right hand side. On the first line, the `reset` that delimits the captured context is preserved after the capture, so the context from a single `reset` outward is protected from manipulation by any number of dynamically enclosed `shift` invocations. Informally speaking, `reset` makes any piece of code appear pure to the outside, that is, devoid of control effects. On the second line, the captured context is surrounded by `reset`, so `f` is bound to a pure function.

Neither occurrence of `reset` on the right hand side of (3) is accidental; they are necessary for the operational semantics to match the transform in Figure 1. Despite the appeal of this match, many other delimited control operators have been proposed (historically, both before and after Danvy and Filinski's work) that remove one or both occurrences of `reset` on the right hand side of (3). Three such variations on `shift` are possible, namely `control`, `shift0`, and `control0` below.

$$M[\texttt{(reset}\ C[\texttt{(control f}\ E\texttt{)}]\texttt{)}] \rhd M[\texttt{(reset}\ E'\texttt{)}] \quad \text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (x)}\ C[\texttt{x}]\texttt{)}\} \qquad (8)$$

$$M[\texttt{(reset}\ C[\texttt{(shift0 f}\ E\texttt{)}]\texttt{)}] \rhd M[E'] \quad \text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (x) (reset}\ C[\texttt{x}]\texttt{))}\} \qquad (9)$$

$$M[\texttt{(reset}\ C[\texttt{(control0 f}\ E\texttt{)}]\texttt{)}] \rhd M[E'] \quad \text{where } E' = E\{\texttt{f} \mapsto \texttt{(lambda (x)}\ C[\texttt{x}]\texttt{)}\} \qquad (10)$$

Felleisen's `control` operator [18, 19, 21, 22, 49], the first delimited control operator in the literature, captures a delimited context without surrounding it with `reset`, so `f` may operate on the contexts in which it is subsequently invoked. The difference between `shift` and `control` can be observed as follows: the program

```
(reset (let ((y (shift f (cons 'a (f '())))))
         (shift g y)))
```

evaluates to `(a)`,[5] whereas the program

```
(reset (let ((y (control f (cons 'a (f '())))))
         (control g y)))
```

evaluates to `()`.[6] Sitaram's `fcontrol` [48] is closely related to `control` in nature. These authors refer to `reset` as `prompt`, `run`, `#`, or `%`.

The `shift0` operator captures a delimited context like `shift` does, but removes the delimiting `reset`. For example, the program

```
(reset (cons 'a
         (reset (shift f (shift g '())))))
```

evaluates to `(a)`,[7] whereas the program

```
(reset (cons 'a
         (reset (shift0 f (shift0 g '())))))
```

evaluates to `()`.[8] Danvy and Filinski [11] consider this `shift0` operator briefly. Also, Hieb and Dybvig's `spawn` [32] can be thought of as a `reset` that, each time it is invoked to insert a new delimiter, creates a specific `shift0` operator for that new delimiter.

The `control0` operator is like `control` but removes the delimit-

[5] The reduction sequence begins:
```
(reset (cons 'a
         ((lambda (x)
            (reset (let ((y x)) (shift g y))))
          '())))
(reset (cons 'a
         (reset (let ((y '())) (shift g y)))))
(reset (cons 'a (reset (shift g '()))))
(reset (cons 'a (reset '())))
```
Here `shift f` introduces a `reset` under the `lambda`, which stops `shift g` from capturing `cons 'a`.

[6] The reduction sequence begins:
```
(reset (cons 'a
         ((lambda (x)
            (let ((y x)) (control g y)))
          '())))
(reset (cons 'a
         (let ((y '())) (control g y))))
(reset (cons 'a (control g '())))
(reset '())
```
Here `control f` allows `control g` to capture `cons 'a`.

[7] The reduction sequence begins:
```
(reset (cons 'a (reset (shift g '()))))
(reset (cons 'a (reset '())))
```

[8] The reduction sequence is:
```
(reset (cons 'a (shift0 g '())))
'()
```

ing `reset`. It is essentially Gunter et al.'s `cupto` [30, 31] stripped down to one prompt variable, and closely related to Queinnec and Serpette's `splitter` [45].

Described operationally as in (8–10), these variations on `shift` seem like minor changes with little sense of purpose. Because adding `reset` is easy, `control` and `shift0` can obviously macro-express `shift`, and `control0` can macro-express them all, without `call/cc` or mutable state. The opposite direction—whether `shift` can simulate any of its `reset`-removed cousins, for example—"seems not to be known" to Gunter et al. [30, 31]. Since no version of `shift` is clearly "right", Gunter et al. choose to take `control0` as primitive.

Concomitant with the apparent difficulty of using `shift` to simulate the other control operators is an apparent difficulty of devising denotational semantics for these operators under a standard CPS transform. More precisely, unlike with `shift`, it is unclear how to translate `control`, `shift0`, or `control0` away using a transform that coincides on pure λ-terms with the first three lines of Figure 1, where contexts are represented as continuation functions. Instead, semantics for these operators in the literature either rely on complex mutable data structures (in essence defining the operators by implementing them in Scheme) or represent contexts as sequences of activation frames,[9] termed *abstract continuations* [21, 22, 42]. Standard continuation semantics is declared "inadequate" [21] and "insufficient" [22],[10] as `control` is said to "admit no such simple static interpretation" [13]. Such claims are surprising in hindsight of Filinski's representation of monads in terms of `shift` and `reset` [23–25]—surely even including `control0` in a language would not disqualify it from Moggi's notions of computation [40]?

Danvy and Filinski [11–13] informally classify their `shift` and `reset` operators as *lexical* and *static*, and other delimited control operators such as `control` as *dynamic*. They use these words to draw an analogy to lexical versus dynamic scoping for variables: roughly speaking, `shift` and `reset`, unlike the other operators, can be defined and implemented without traversing arbitrarily deeply into data structures at run-time. The next section shows that, as soon as we allow traversing arbitrarily deeply into data structures at run-time, dynamic control operators can be treated with the same transform as static ones. That is, continuation semantics is sufficient after all, as long as the continuation can be recursive.[11]

Our development below of recursive continuations is guided by recursive types. For example, if α is a type, then the type $\mathsf{List}\ \alpha$ of singly-linked α-lists can be defined by

$$\mathsf{List}\ \alpha \;=\; 1 + \alpha \times \mathsf{List}\ \alpha, \tag{11}$$

where 1 is the unit type and $\times$ constructs product types. For brevity, we take the unfolding of a recursive type to give not just isomorphic but in fact equivalent types. For example, (11) states an equation between types, not just an isomorphism. To use terms coined by Crary et al. [6, 27], this paper shows *equi-recursive* types, but *iso-recursive* types can be used too.

[9] Or an algebra thereof.

[10] A reviewer states that these declarations are objections to the non-tail calls in Figure 1 (as continuation semantics for `shift` and `reset`) and (18–19) in Section 3.1 (as continuation semantics for `control` and `prompt`). However, see footnote 4.

[11] One way to see the connection between dynamic control operators and recursive continuations may be to observe how the following program enters an infinite loop.

```
(prompt (begin (control f (begin (f 0) (f 0)))
               (control f (begin (f 0) (f 0)))))
```

3 Recursive continuations

In this central section of the paper, we treat dynamic control operators by extending the standard CPS transform, and by translating them into `shift` and `reset`. The key to these treatments is to represent delimited contexts as functions whose types are recursive: When a delimited context is captured with a dynamic control operator, then invoked, it may take control over the delimited context at the invocation site. Hence, the former context must take the latter context as an argument in our CPS transform. Roughly speaking, then, the type of contexts must mention itself, that is, be recursive.

Let us first review delimited contexts captured by `shift` and `reset`. The CPS transform in Figure 1 represents a delimited context as a continuation, that is, a function of type $\tau \to \omega$. Danvy and Filinski identify τ with the type of the intermediate result (that is, the hole in the context) and ω with the type of the answer (that is, the context once plugged). For comparison with other control operators below, we define the types

$$\mathsf{Context}\ \tau\ \omega \;=\; \tau \to \omega, \tag{12}$$

$$\mathsf{Answer}\ \omega \;=\; \omega, \tag{13}$$

such that

$$\mathsf{Context}\ \tau\ \omega \;=\; \tau \to \mathsf{Answer}\ \omega. \tag{14}$$

To take an example, the delimited context `[(< 1 _)]` takes the type $\mathsf{Context}\ \mathsf{Int}\ \mathsf{Bool}$ (or equivalently, $\mathsf{Int} \to \mathsf{Bool}$) when captured with `shift`, because plugging the hole `_` with an integer gives an answer that is a boolean. In other words, the function

```
(lambda (x)
  (reset (< 1 x)))
```

(which represents that context, as captured by `shift`) maps integers to booleans. For another example, the delimited context `[(let ((y _)) (shift g (< 1 y)))]`, when captured by `shift`, also has the type $\mathsf{Context}\ \mathsf{Int}\ \mathsf{Bool}$. In other words, the function

```
(lambda (x)
  (reset (let ((y x)) (shift g (< 1 y)))))
```

(which represents that context, as captured by `shift`) also maps integers to booleans. In fact, these two contexts captured by `shift` are observationally equivalent, because the `shift g` above has only the empty delimited context `[_]` to capture.

3.1 `control`

The context `[(let ((y _)) (control g (< 1 y)))]` captured with `control` is not equivalent to `[(< 1 _)]`, because the function

```
(lambda (x)
  (let ((y x)) (control g (< 1 y))))
```

(which represents the first context, as captured by `control`) wipes out its surrounding delimited context when invoked, whereas the function

```
(lambda (x)
  (< 1 x))
```

(which represents the second context, as captured by `control`) does not. In general, when a delimited context captured by `control` is invoked, it may further capture the surrounding delimited context (up to the nearest dynamically enclosing `reset`) at the point of invocation. Thus a delimited context captured by `control`, unlike one captured by `shift`, is not a function from an intermediate result (with which to plug a hole) to a final answer. Rather, a `control`-captured context can be thought of as a function from an intermediate result *and any surrounding delimited context* to a final answer. The surrounding context may be the empty context $[\_]$ (if the captured context is invoked immediately within `reset`) or not empty. Accordingly, we let a delimited context captured by `control` whose hole is of type τ and answer is of type ω take the type $\mathsf{Context}'\,\tau\,\omega$, where

$$\mathsf{Context}'\,\tau\,\omega \;=\; \tau \to \mathsf{Maybe}\,(\mathsf{Context}'\,\omega\,\omega) \to \omega. \qquad (15)$$

In this recursive type definition, $\mathsf{Maybe}\,\alpha$ means either an α-value or the special token `#f`, like the discriminated union types `Maybe a` in Haskell. We use `#f` to represent the empty surrounding context.

The function `send` below plugs an intermediate answer `v` (of type ω) into a delimited context `mc` (of type $\mathsf{Maybe}\,(\mathsf{Context}'\,\omega\,\omega)$) by calling `mc` with `v` and the trivial delimited context `#f`. If `mc` is the special token `#f`, then we are plugging `v` into the empty context, so the final answer is just `v`.

```
(define (send v)
  (lambda (mc) (if mc ((mc v) #f) v)))
```

This function is of type $\mathsf{Context}'\,\omega\,\omega$: it is itself a delimited context, namely the empty one. If our target language lets us compare values against `send` (even intensionally using `eq?`, say), then we can do so rather than comparing values against `#f`, and drop our use of Maybe. That is, we could implement `send` as

```
(define (send v)
  (lambda (mc)
    (if (eq? send mc) v ((mc v) send))))
```

but do not, for clarity.

When two `shift`-captured contexts are composed as functions at the source level, the result corresponds to concatenating continuations by function composition in the target language. By contrast, to concatenate `control`-captured contexts of the recursive type defined in (15), we define a recursive function, of type $\big(\mathsf{Context}'\,\tau\,\omega \times \mathsf{Maybe}\,(\mathsf{Context}'\,\omega\,\omega)\big) \to \mathsf{Context}'\,\tau\,\omega$:

```
(define (compose c mc1)
  (if mc1 (lambda (v)
            (lambda (mc2)
              ((c v) (compose mc1 mc2))))
          c))
```

According to (15), the type $\mathsf{Context}'\,\tau\,\omega$ is a function type, and τ only appears in its domain, not codomain. In other words, a context captured by `control` whose hole type is τ has the function type of a τ-continuation, just like delimited contexts captured by `shift`, except for the recursive answer type $\mathsf{Answer}'\,\omega$ defined by

$$\begin{aligned}\mathsf{Answer}'\,\omega \;&=\; \mathsf{Maybe}\,(\mathsf{Context}'\,\omega\,\omega) \to \omega \\ &=\; \mathsf{Maybe}\,(\omega \to \mathsf{Answer}'\,\omega) \to \omega, \end{aligned} \qquad (16)$$

such that

$$\begin{aligned}\mathsf{Context}'\,\tau\,\omega \;&=\; \tau \to \mathsf{Answer}'\,\omega \\ &=\; \mathsf{Context}\,\tau\,(\mathsf{Answer}'\,\omega). \end{aligned} \qquad (17)$$

Thus $\mathsf{Context}'$ can be written in terms of Context! Hence, delimited contexts captured by `control` can be represented as ordinary, if recursive, continuations. The equations below extend the first three lines of Figure 1 to `control`. It maps every source type τ, at the top level or to the right of a function arrow, to a type of the form $(\tau \to \mathsf{Answer}'\,\omega) \to \mathsf{Answer}'\,\omega$. To distinguish the `reset` for `control` here from the `reset` for `shift` above, we write `prompt` instead of `reset`.

$$\overline{\texttt{(prompt } E\texttt{)}} = \texttt{(lambda (c) (c ((}\overline{E}\texttt{ send) #f)))} \qquad (18)$$

$$\begin{aligned}&\overline{\texttt{(control f } E\texttt{)}} = \\ &\texttt{(lambda (c1)}\\ &\quad\texttt{(lambda (mc1)}\\ &\quad\quad\texttt{(let ((f (lambda (x)}\\ &\quad\quad\quad\quad\texttt{(lambda (c2)}\\ &\quad\quad\quad\quad\quad\texttt{(lambda (mc2)}\\ &\quad\quad\quad\quad\quad\quad\texttt{(((compose c1 mc1) x)}\\ &\quad\quad\quad\quad\quad\quad\texttt{(compose c2 mc2)))))))}\\ &\quad\quad\texttt{((}\overline{E}\texttt{ send) #f))))}\end{aligned} \qquad (19)$$

Because this transform extends a standard call-by-value CPS transform on the pure λ-calculus, it shows how to treat `control` and `prompt` as operations in the continuation monad (with answer type $\mathsf{Answer}'\,\omega$). Then, because `shift` and `reset` expresses all operations in the continuation monad, we can define `control` and `prompt` in direct style as macros in terms of `shift` and `reset`.

```
(define-syntax prompt
  (syntax-rules ()
    ((_ e) ((reset (send e)) #f))))

(define-syntax control
  (syntax-rules ()
    ((_ f e)
     (shift c1
       (lambda (mc1)
         (let ((f (lambda (x)
                    (shift c2
                      (lambda (mc2)
                        (((compose c1 mc1) x)
                         (compose c2 mc2)))))))
           ((reset (send e)) #f)))))))
```

These source-level macros correspond directly to the target-level equations (18–19), except:

- Where the target-level equations abstract over a continuation argument, the source-level macros use `shift` rather than `lambda`.
- Where the equations pass the continuation `send` to $\overline{E}$, the macros say `(reset (send` E`))`, so as to place E in the delimited context `[(send _)]`.

This implementation of `control` and `prompt` uses neither `call/cc` nor mutable state; in particular, it does not capture any continuation beyond the outermost delimiting `prompt`.

Another way to view the same definitions in hindsight is to recognize that a denotational semantics given by Felleisen et al. [21, Section 4] encodes `control` and `prompt` in a monad that maps each type τ to the type $(\tau \to \mathsf{Answer}'\,\omega) \to \omega$. This monad is not the continuation monad, because the answer types $\mathsf{Answer}'\,\omega$ and ω are different; hence, Felleisen et al.'s equations for their denotational semantics do not give a standard CPS transform. Nevertheless, we

can still use Filinski's representation of monads in terms of `shift` and `reset` [23–25] to represent `control` and `prompt`—essentially as above, in fact. As an anonymous reviewer hints, this observation is one way to show our definitions to correctly implement `control` and `prompt`.

Sitaram and Felleisen [49] implement `control` and `prompt` in terms of `call/cc` in Scheme. That implementation uses both `call/cc` and mutable state. Our implementation of `control` and `prompt` using `shift` and `reset` can be composed with Filinski's implementation of `shift` and `reset` using `call/cc` [23] to yield a more modular implementation of `control` and `prompt` using `call/cc`. Sitaram and Felleisen's implementation maintains a global, mutable *run-stack*. The run-stack is comprised of *sub-stacks*, one for each dynamically active `prompt`. Each sub-stack is a list of invocation points (that is, undelimited continuations captured by `call/cc`). These data structures can be correlated with our implementation: The run-stack is a sequence of "`mc`" functions (of type $\mathsf{Maybe}\,(\mathsf{Context}'\ \omega\ \omega)$), one for each dynamically active `prompt`. Each `mc` function is a sub-stack, the result of concatenating `control`-captured contexts using `compose`.

3.2 `shift0`

When `shift0` captures a delimited context, it does not replace it with the trivial delimited context as `shift` does. Instead, it removes the captured context along with its delimiting `reset`, exposing the next-outer delimited context up to the next-nearest dynamically enclosing `reset`. With `shift0` in the language, `reset` is not idempotent: `(reset` E`)` is not equivalent to `(reset (reset` E`))`, because each `reset` only "defends against" one `shift0`. For example, the program

```
(reset (cons 'a
         (reset (shift0 f (shift0 g '())))))
```

evaluates to `()`, but the program

```
(reset (cons 'a
         (reset
           (reset (shift0 f (shift0 g '()))))))
```

evaluates to `(a)`.

Because `shift0` removes the delimiting `reset` when capturing a delimited context, the context

```
[(let ((y _))
   (shift0 f (shift0 g (< 1 y))))]
```

captured with `shift0` is not equivalent to the contexts

```
[(let ((y _)) (shift0 g (< 1 y)))]
[(< 1 _)]
```

captured with `shift0`. That is, the function

```
(lambda (x)
  (reset (let ((y x))
           (shift0 f (shift0 g (< 1 y))))))
```

wipes out its surrounding delimited context when invoked, whereas the functions

```
(lambda (x)
  (reset (let ((y x)) (shift0 g (< 1 y)))))
(lambda (x)
  (reset (< 1 x)))
```

do not.

Appendix C of Danvy and Filinski's technical report [11] considers this variation on `shift` briefly. They model it denotationally by passing around a list of delimited contexts, which can be thought of as a sequence of activation frames, except each frame corresponds to a `reset` rather than a function call.[12] In our formulation, a delimited context captured by `shift0` whose hole type is τ and whose answer type is ω has the type $\mathsf{Context}_0\ \tau\ \omega$, where

$$\mathsf{Context}_0\ \tau\ \omega \;=\; \tau \to \mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega) \to \omega. \qquad (20)$$

In this recursive type definition, $\mathsf{List}\ \alpha$ means a singly-linked list of α-values, either a `cons` cell or the empty list `()`. A list of type $\mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega)$ contains delimited contexts from innermost to outermost, separated by control delimiters.

The function `propagate` below plugs an intermediate answer `v` (of type ω) into a list of contexts `lc` (of type $\mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega)$) by calling the head of `lc` with `v` and the tail of `lc`. If `c` is empty, then the final answer is simply `v`.

```
(define (propagate v)
  (lambda (lc)
    (if (null? lc) v
                   (((car lc) v) (cdr lc)))))
```

This function is of type $\mathsf{Context}_0\ \omega\ \omega$: it is itself a delimited context, namely the empty one.

Like the type $\mathsf{Context}'\ \tau\ \omega$ in Section 3.1, $\mathsf{Context}_0\ \tau\ \omega$ is a function type in which τ only appears in the domain. Hence a delimited context captured by `shift0` is just like one captured by `shift`, except the answer type $\mathsf{Answer}_0\ \omega$ of the continuation is recursive, defined by

$$\begin{aligned}\mathsf{Answer}_0\ \omega \;&=\; \mathsf{List}\,(\mathsf{Context}_0\ \omega\ \omega) \to \omega \\ &=\; \mathsf{List}\,(\omega \to \mathsf{Answer}_0\ \omega) \to \omega, \end{aligned} \qquad (21)$$

such that

$$\begin{aligned}\mathsf{Context}_0\ \tau\ \omega \;&=\; \tau \to \mathsf{Answer}_0\ \omega \\ &=\; \mathsf{Context}\ \tau\ (\mathsf{Answer}_0\ \omega). \end{aligned} \qquad (22)$$

Thus $\mathsf{Context}_0$ can be written in terms of $\mathsf{Context}$. Therefore, just as with `control`, delimited contexts captured by `shift0` can be represented as ordinary continuations. Following the Appendix C mentioned above, the equations below extend the first three lines of Figure 1 to a CPS transform for `shift0`. It maps every source type τ, at the top level or to the right of a function arrow, to a type of the form $(\tau \to \mathsf{Answer}_0\ \omega) \to \mathsf{Answer}_0\ \omega$. To distinguish the `reset` for `shift0` here from the `reset` for `shift` above, we write `reset0` instead of `reset`.

$$\overline{\texttt{(reset0}\ E\texttt{)}} =$$
```
  (lambda (c)
    (lambda (lc)
      ((E-bar propagate) (cons c lc))))
```
(23)

[12] Johnson and Duggan [34] add control facilities to the programming language GL that provide power similar to that of `shift0` and `reset`, but they make each function call delimit the context (like Landin's SECD machine [9, 10, 37]), so their frames do correspond to function calls.

```
(shift0 f E) =
    (lambda (c1)
      (lambda (lc)
        (let ((f (lambda (x)
                   (lambda (c2)
                     (lambda (lc)
                       ((c1 x) (cons c2 lc)))))))
          ((E (car lc)) (cdr lc)))))                    (24)
```

As in Section 3.1, these equations[13] can be turned into a direct implementation of `shift0` and `reset0` in terms of `shift` and `reset` that neither captures undelimited continuations nor keeps mutable state.

3.3 control0

The `control0` operator removes both occurrences of `reset` on the right hand side of (3); it combines the dynamic properties of `control` and `shift0`. It is thus not surprising that we can treat `control0` with recursive continuations and the CPS transform by combining the ideas from Sections 3.1–2.

A delimited context captured by `control0`, with hole type τ and answer type ω, has the type

$$\mathsf{Context}'_0\,\tau\,\omega \;=\; \tau \to \mathsf{Maybe}\,(\mathsf{Context}'_0\,\omega\,\omega) \to \mathsf{List}\,(\mathsf{Context}'_0\,\omega\,\omega) \to \omega, \qquad (25)$$

in which τ only appears in the domain. A delimited context captured by `control0` is thus just like one captured by `shift` with the recursive answer type

$$\begin{aligned}\mathsf{Answer}'_0\,\omega \;&=\; \mathsf{Maybe}\,(\mathsf{Context}'_0\,\omega\,\omega) \to \mathsf{List}\,(\mathsf{Context}'_0\,\omega\,\omega) \to \omega \\ &=\; \mathsf{Maybe}\,(\omega \to \mathsf{Answer}'_0\,\omega) \to \mathsf{List}\,(\omega \to \mathsf{Answer}'_0\,\omega) \to \omega, \end{aligned} \qquad (26)$$

such that

$$\begin{aligned}\mathsf{Context}'_0\,\tau\,\omega \;&=\; \tau \to \mathsf{Answer}'_0\,\omega \\ &=\; \mathsf{Context}\,\tau\,(\mathsf{Answer}'_0\,\omega). \end{aligned} \qquad (27)$$

Thus $\mathsf{Context}'_0$ can be written in terms of Context. Informally speaking, the Maybe part of the types above keeps track of the delimited context within the nearest dynamically enclosing `reset`, and the List part keeps track of the delimited contexts beyond that `reset`.

The trivial delimited context of type $\mathsf{Context}'_0\,\omega\,\omega$ is the function `send-propagate` below, which combines `send` and `propagate`.

```
(define (send-propagate v)
  (lambda (mc)
    (if mc ((mc v) #f)
           (lambda (lc)
             (if (null? lc) v
                            ((((car lc) v) #f)
                             (cdr lc)))))))
```

To compose delimited contexts captured by `control0`, we can simply use the code for `compose` above, because—although it is created for `control`—it also has the type

$$\big(\mathsf{Context}'_0\,\tau\,\omega \times \mathsf{Maybe}\,(\mathsf{Context}'_0\,\omega\,\omega)\big) \to \mathsf{Context}'_0\,\tau\,\omega. \qquad (28)$$

Finally, we can use `send-propagate` and `compose` to define an ordinary CPS transform for `control0`. Here we write `prompt0` instead of `reset` to mean the `reset` for `control0`.

```
(prompt0 E) =
  (lambda (c)
    (lambda (mc)
      (lambda (lc)
        (((E send-propagate) #f)
         (cons (compose c mc) lc)))))                   (29)

(control0 f E) =
    (lambda (c1)
      (lambda (mc1)
        (lambda (lc)
          (let ((f (lambda (x)
                     (lambda (c2)
                       (lambda (mc2)
                         (((compose c1 mc1) x)
                          (compose c2 mc2)))))))
            (((E (car lc)) #f) (cdr lc)))))))           (30)
```

This CPS transform maps every source type τ, at the top level or to the right of a function arrow, to a type of the form $(\tau \to \mathsf{Answer}'_0\,\omega) \to \mathsf{Answer}'_0\,\omega$. Again, these CPS equations can be turned into an implementation of `control0` and `prompt0` using `shift` and `reset` that neither captures undelimited continuations nor keeps mutable state.

4 Conclusion and related work

This paper presents the first CPS transform for dynamic delimited control operators, including Felleisen's `control` and `prompt`, that is consistent with a standard CPS transform. We have shown that Danvy and Filinski's static operators `shift` and `reset` are just as expressive as dynamic ones. For a delimited control operator to be dynamic is for it to require recursive continuations.

Now that we know how to implement dynamic operators in terms of `shift` and `reset` without capturing undelimited continuations or keeping mutable state, direct implementations of `shift` and `reset` like Gasbichler and Sperber's [28] give rise to direct implementations of dynamic operators. Moreover, because our CPS transform extends a standard one, it can be incorporated into CPS-based language implementations.

Besides explicating dynamic control operators, recursive continuations are also useful in practical programming. For example, the iterative interaction pattern between a coroutine and its environment is reflected in a recursive continuation, specifically its recursive answer type [25, Section 4.2], which can be depicted graphically as a flowchart. Two special cases of such interactions are:

- the interaction between a Web server and user agents [16, 29, 43, 44]; and
- the interaction between a cursor iterating over a collection and its client [36], as epitomized in the classic same-fringe problem.

Another potential application of recursive continuations lies in Balat et al.'s type-directed partial evaluator for the λ-calculus with products and sums [2], which computes normal forms for λ-terms

[13] Now in CPS; see footnote 4. Expressions like `((E propagate) (cons c lc))` may appear to contain a non-tail call, but should be regarded as a curried call with two arguments.

under βη-equivalence. To normalize terms that use sums, Balat et al.'s algorithm uses Gunter et al.'s `cupto` operator [30, 31], rather than `shift` as in previous work by Balat and Danvy [1]. As Balat et al.'s algorithm evaluates a term, it keeps a list of possible scope locations at which future `case` expressions may be inserted, in the form of prompts for `cupto`. (By contrast, Balat and Danvy's earlier algorithm using `shift` only considers one scope location at which to insert a `case` expression.) If `cupto` is replaced by `shift` with a recursive continuation, then that list of prompts would be pleasingly identified with the stack of control points that Gunter et al. use to implement `cupto` in the first place. A direct implementation of `cupto` or `shift` would also make the algorithm more efficient.

5 Acknowledgements

This paper would not be written without the help and encouragement of Oleg Kiselyov. Thanks also to Chris Barker, John Clements, Olivier Danvy, Matthias Felleisen, Andrzej Filinski, Shriram Krishnamurthi, Stuart Shieber, Sam Tobin-Hochstadt, and six anonymous reviewers for ICFP 2004 and this workshop. This work is supported by the United States National Science Foundation Grant BCS-0236592.

References

[1] Balat, Vincent, and Olivier Danvy. 2002. Memoization in type-directed partial evaluation. In *Proceedings of GPCE 2002: 1st ACM conference on generative programming and component engineering*, ed. Don S. Batory, Charles Consel, and Walid Taha, 78–92. Lecture Notes in Computer Science 2487, Berlin: Springer-Verlag.

[2] Balat, Vincent, Roberto Di Cosmo, and Marcelo Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL '04: Conference record of the annual ACM symposium on principles of programming languages*, 64–76. New York: ACM Press.

[3] Barker, Chris. 2004. Continuations in natural language (extended abstract). In [51], 1–11.

[4] Biernacka, Małgorzata, Dariusz Biernacki, and Olivier Danvy. 2004. An operational foundation for delimited continuations. In [51], 25–33.

[5] Clinger, William D. 1998. Proper tail recursion and space efficiency. In *POPL '98: Conference record of the annual ACM symposium on principles of programming languages*, 174–185. New York: ACM Press.

[6] Crary, Karl, Robert Harper, and Sidd Puri. 1999. What is a recursive module? In *PLDI '99: Proceedings of the ACM conference on programming language design and implementation*, vol. 34(5) of *ACM SIGPLAN Notices*, 50–63. New York: ACM Press.

[7] Danvy, Olivier. 1994. Back to direct style. *Science of Computer Programming* 22(3):183–195.

[8] ———. 1996. Type-directed partial evaluation. In *POPL '96: Conference record of the annual ACM symposium on principles of programming languages*, 242–257. New York: ACM Press.

[9] ———. 2003. A rational deconstruction of Landin's SECD machine. Report RS-03-33, BRICS, Denmark.

[10] ———. 2004. On evaluation contexts, continuations, and the rest of the computation. In [51].

[11] Danvy, Olivier, and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark. `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz`.

[12] ———. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.

[13] ———. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

[14] Danvy, Olivier, and Julia L. Lawall. 1992. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM conference on Lisp and functional programming*, ed. William D. Clinger, vol. V(1) of *Lisp Pointers*, 299–310. New York: ACM Press.

[15] ———. 1996. Back to direct style II: First-class continuations. Report RS-96-20, BRICS, Denmark.

[16] Double, Chris. 2004. Partial continuations. `http://www.double.co.nz/scheme/partial-continuations/partial-continuations.html`.

[17] Dybjer, Peter, and Andrzej Filinski. 2002. Normalization and partial evaluation. In *APPSEM 2000: International summer school on applied semantics, advanced lectures*, ed. Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, 137–192. Lecture Notes in Computer Science 2395, Berlin: Springer-Verlag.

[18] Felleisen, Matthias. 1987. The calculi of λ_v-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Indiana University. Also as Tech. Rep. 226, Department of Computer Science, Indiana University.

[19] ———. 1988. The theory and practice of first-class prompts. In [41], 180–190.

[20] ———. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17(1–3):35–75.

[21] Felleisen, Matthias, Daniel P. Friedman, Bruce F. Duba, and John Merrill. 1987. Beyond continuations. Tech. Rep. 216, Computer Science Department, Indiana University.

[22] Felleisen, Matthias, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. 1988. Abstract continuations: A mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM conference on Lisp and functional programming*, 52–62. New York: ACM Press.

[23] Filinski, Andrzej. 1994. Representing monads. In *POPL '94: Conference record of the annual ACM symposium on principles of programming languages*, 446–457. New York: ACM Press.

[24] ———. 1996. Controlling effects. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Also as Tech. Rep. CMU-CS-96-119.

[25] ———. 1999. Representing layered monads. In *POPL '99: Conference record of the annual ACM symposium on principles of programming languages*, 175–188. New York: ACM Press.

[26] ———. 2001. Normalization by evaluation for the compuational lambda-calculus. In *TLCA 2001: Proceedings of the 5th international conference on typed lambda calculi and applications*, ed. Samson Abramsky, 151–165. Lecture Notes in Computer Science 2044, Berlin: Springer-Verlag.

[27] Gapeyev, Vladimir, Michael Y. Levin, and Benjamin C. Pierce. 2000. Recursive subtyping revealed. In [33], 221–231.

[28] Gasbichler, Martin, and Michael Sperber. 2002. Final shift for call/cc: Direct implementation of shift and reset. In *ICFP '02: Proceedings of the ACM international conference on functional programming*, 271–282. New York: ACM Press.

[29] Graunke, Paul Thorsen. 2003. Web interactions. Ph.D. thesis, College of Computer Science, Northeastern University.

[30] Gunter, Carl A., Didier Rémy, and Jon G. Riecke. 1995. A generalization of exceptions and control in ML-like languages. In *Functional programming languages and computer architecture: 7th conference*, ed. Simon L. Peyton Jones, 12–23. New York: ACM Press.

[31] ———. 1998. Return types for functional continuations. `http://pauillac.inria.fr/~remy/work/cupto/`.

[32] Hieb, Robert, and R. Kent Dybvig. 1990. Continuations and concurrency. In *Proceedings of the 2nd ACM SIGPLAN symposium on principles and practice of parallel programming*, 128–136. New York: ACM Press.

[33] ICFP. 2000. *ICFP '00: Proceedings of the ACM international conference on functional programming*, vol. 35(9) of *ACM SIGPLAN Notices*. New York: ACM Press.

[34] Johnson, Gregory F., and Dominic Duggan. 1988. Stores and partial continuations as first-class objects in a language and its environment. In [41], 158–168.

[35] Kelsey, Richard, William D. Clinger, Jonathan Rees, Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, G. J. Rozas, N. I. Adams, IV, Daniel P. Friedman, Eugene Kohlbecker, Guy L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and Mitchell Wand. 1998. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation* 11(1):7–105. Also as *ACM SIGPLAN Notices* 33(9):26–76.

[36] Kiselyov, Oleg. 2004. General ways to traverse collections. `http://okmij.org/ftp/Scheme/enumerators-callcc.html`.

[37] Landin, Peter J. 1964. The mechanical evaluation of expressions. *The Computer Journal* 6(4):308–320.

[38] Lawall, Julia L., and Olivier Danvy. 1994. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM conference on Lisp and functional programming*, 227–238. New York: ACM Press.

[39] Meyer, Albert R., and Mitchell Wand. 1985. Continuation semantics in typed lambda-calculi (summary). In *Logics of programs*, ed. Rohit Parikh, 219–224. Lecture Notes in Computer Science 193, Berlin: Springer-Verlag.

[40] Moggi, Eugenio. 1991. Notions of computation and monads. *Information and Computation* 93(1):55–92.

[41] POPL. 1988. *POPL '88: Conference record of the annual ACM symposium on principles of programming languages*. New York: ACM Press.

[42] Queinnec, Christian. 1993. A library of high-level control operators. *Lisp Pointers* 6(4):11–26.

[43] ———. 2000. The influence of browsers on evaluators or, continuations to program web servers. In [33], 23–33.

[44] ———. 2001. Inverting back the inversion of control or, continuations versus page-centric programming. Rapport de Recherche LIP6 2001/007, Laboratoire d'Informatique de Paris 6.

[45] Queinnec, Christian, and Bernard Serpette. 1991. A dynamic extent control operator for partial continuations. In *POPL '91: Conference record of the annual ACM symposium on principles of programming languages*, 174–184. New York: ACM Press.

[46] Sabry, Amr, and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* 6(3–4):289–360.

[47] Shan, Chung-chieh. 2004. Delimited continuations in natural language: Quantification and polarity sensitivity. In [51], 55–64.

[48] Sitaram, Dorai. 1993. Handling control. In *PLDI '93: Proceedings of the ACM conference on programming language design and implementation*, vol. 28(6) of *ACM SIGPLAN Notices*, 147–155. New York: ACM Press.

[49] Sitaram, Dorai, and Matthias Felleisen. 1990. Control delimiters and their hierarchies. *Lisp and Symbolic Computation* 3(1):67–99.

[50] Sumii, Eijiro. 2000. An implementation of transparent migration on standard Scheme. In *Proceedings of the workshop on Scheme and functional programming*, ed. Matthias Felleisen, 61–63. Tech. Rep. 00-368, Department of Computer Science, Rice University.

[51] Thielecke, Hayo, ed. 2004. *CW'04: Proceedings of the 4th ACM SIGPLAN workshop on continuations*. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.

[52] Thiemann, Peter. 1999. Combinators for program generation. *Journal of Functional Programming* 9(5):483–525.

Cleaning up the Tower: Numbers in Scheme

Sebastian Egner
Philips Research Laboratories
sebastian.egner@philips.com

Richard A. Kelsey
Ember Corporation
kelsey@s48.org

Michael Sperber
sperber@deinprogramm.de

Abstract

The R^{5}RS specification of numerical operations leads to unportable and intransparent behavior of programs. Specifically, the notion of "exact/inexact numbers" and the misleading distinction between "real" and "rational" numbers are two primary sources of confusion. Consequently, the way R^{5}RS organizes numbers is significantly less useful than it could be. Based on this diagnosis, we propose to abandon the concept of exact/inexact numbers from Scheme altogether. In this paper, we examine designs in which exact and inexact rounding *operations* are explicitly separated, while there is no distinction between exact and inexact numbers. Through examining alternatives and practical ramifications, we arrive at an alternative proposal for the design of the numerical operations in Scheme.

1 Introduction

The set of numerical operations of a wide-spectrum programming language ideally satisfies the following requirements:

efficiency The programming language's operations are reasonably efficient relative to the capabilities of the underlying machine. In practice, this means that a program can employ fixnum and floating-point arithmetic where reduced precision is acceptable.

accuracy A program computes with numbers without introducing error.

reproducibility The same program, run on different language implementations, will produce the same result.

transparency The programmer can tell when a result is the outcome of inexact operations and thus contains error, or when a computation is reproducible exactly.

In practice, efficiency and accuracy are often in conflict: Accurate computations on non-integral numbers are often (but not always) prohibitively expensive. Fast floating-point arithmetic introduces error. Thus, a realistic programming language must choose a compromise between the two—which introduces the need for transparency. Reproducibility is clearly desirable, but also often in conflict with efficiency—the most efficient method for performing a computation on one machine may be inefficient on another. At least, a programmer should be able to predict whether a program computes a reproducible result. Moreover, as many practical programs as possible should in fact run reproducibly.

Scheme 48 1.1 (default mode)	`7/10`
Petite Chez Scheme 6.0a, Gambit-C 3.0, Scheme 48 1.1 (after `,open floatnums`)	`3152519739159347/4503599627370496`
SCM 5d9	`1`
Chicken 0/1082:	"can not be represented as an exact number"

Table 1. Value of `(inexact->exact 0.7)` in various R^{5}RS Scheme implementations

R^{5}RS [13] provides for *exact* and *inexact* numbers, with the idea that operations on exact numbers are accurate but potentially inefficient and operations on inexact numbers are efficient but introduce error. The intention behind R^{5}RS is to hide the actual machine representation of numbers, and to allow the program (or the programmer) to look at a number object and determine whether it contains error. In theory, this would fulfill a reasonable transparency requirement. In practice, however, the numerical operations in Scheme are anything but transparent.

For a trivial example exhibiting the poor reproducibility of R^{5}RS programs, consider the value of the expression `(inexact->exact 0.7)` in various Scheme systems, all of which are completely R^{5}RS-compliant. Table 1 shows that the results vary wildly—Scheme 48 can even change its behavior at run time. This is only one of a wide variety of problems a programmer faces who tries to predict the outcome of a computation on numbers in a Scheme program. Clearly, R^{5}RS provides for little reproducibility.

What is the cause of these problems? R^{5}RS takes the stance that programs using exact arithmetic are essentially the same as programs using inexact arithmetic. The procedures they use are the same, after all—only the numbers are different. In reality, programs using inexact arithmetic operations are inherently different from programs using only exact operations. By blurring the distinction, R^{5}RS complicates writing many programs dealing with numbers. We know of no design approach which does successfully unite exact and inexact arithmetic into a common set of operations without sacrificing transparency and reproducibility.

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

In this paper, we examine the specific problems with the numerical operations specified in R^{5}RS, and consider alternative designs. Specifically, our contributions are the following:

- We identify the design fallacies concerning the numerical operations specified in R^{5}RS, and their practical consequences.
- We show how to design numerical operations around the idea that *operations* rather than *numbers* are exact or inexact. The design has the following properties:
  - It uses a different numerical tower that is a more appropriate model for realistic program, and which a Scheme system can realistically represent accurately.
  - All standard numerical operations operate exactly on rational numbers of infinite precision.
  - The floating-point operations are separate procedures.
- We examine the choices available in such a design, and discuss their consequences and relative merits. The choices concern the relationship between the rational numbers and the floating-point numbers—whether floating-point numbers count as rationals and vice versa. We also discuss the nature of $\pm\infty$ and "not-a-number," and where they fit in our framework.

  All of the design alternatives we examine allow compromises between efficiency and accuracy similar to what R^{5}RS currently provides, as well as improved transparency and reproducibility: Any program that does not contain calls to floating-point operations always computes exactly and reproducibly, independent of the Scheme implementation it runs on. Rounding conversions from rational to floating-point numbers only occur at clearly identifiable places in a program.
- We identify some weaknesses in the R^{5}RS set of numerical operations as they pertain to the new design, and describe possible approaches to addressing them. This includes the definitions of `quotient`, `remainder`, and `modulo`, the definitions of the rounding operations, and dealing with external representations.

We do not address all the issues concerning numbers that a future revision of the Scheme standard should address. Specifically, we do not discuss the relative merits of tying the Scheme standard to a specific floating-point representation. We do not touch the issue of offering abstractions for explicitly controlling their propagation, as well as the rounding mode of floating-point operations: This is addressed in detail elsewhere, for example in the recent work on floating-point arithmetic in Java [4]. Also, we omit complex numbers and other advanced number representations (such as algebraic numbers, interval arithmetic, cyclotomic fields etc.) from the discussion; they are largely orthogonal to the subject of this paper.

Overview We identify the main problems with the R^{5}RS approach in Section 2. In Section 3, we present a new model for exact rational arithmetic and a set of typical machine representations for it. Section 4 describes how to add inexact arithmetic to the model, along with the design issues arising from this. Section 5 explores design alternatives within our model. In Section 6, details a possible set of exact numerical operations. Some implementation issues are discussed in Section 7. Finally, Section 8 lists some related work, and Section 9 concludes.

2 Problems with the R^{5}RS approach

R^{5}RS specifies that the objects representing numbers in a Scheme system must be organized according to a subtype hierarchy called the *numerical tower*:

$$\text{number} \supseteq \text{complex} \supseteq \text{real} \supseteq \text{rational} \supseteq \text{integer}$$

Section 6.2.3 of R^{5}RS requires implementations to provide "a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language." Moreover, implementations may provide "only a limited range of numbers of any type, [...]". As a minimal requirement, exact integers must be present "throughout the range of numbers that may be used for indexes of lists, vectors, and strings [...]".

In addition, Section 6.2.2 specifies that the representation of a number is flagged either exact or inexact and that operations should propagate inexactness as a contagious property. Hence, numbers in R^{5}RS are not just organized according to the numerical tower, but also according to exactness. The exact/inexact distinction is claimed to be "orthogonal to the dimension of type."

The rest of this section enumerates some of the most significant problems with the R^{5}RS specification.

2.1 Not enough numbers

Numbers are in short supply in R^{5}RS. As quoted above, the only numbers a Scheme system must support are indices for arrays, lists and strings. A Scheme system that supports only integers $\{0,\ldots,255\}$ can be perfectly conformant.

Of course, the use of limited-precision fixnum arithmetic can improve performance. However, we conjecture that the cost of allowing the standard arithmetic operations to only support limited precision—the loss of algebraic laws, transparency and reproducibility—is greater than the benefit.

2.2 Unspecified precision of inexact numbers

R^{5}RS puts no constraints on the range or precision of inexact numbers. In particular, the description of `exact->inexact` and `inexact->exact` says

> These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range.

More tellingly, it also says

> If an exact[inexact] argument has no reasonably close inexact[exact] equivalent, then a violation of an implementation restriction may be reported.

The "may" implies that implementations are free to use an arbitrarily inaccurate equivalent. Moreover, the meaning of "reasonably close" is similarly underspecified: Table 1 shows that a call to `inexact->exact` that works fine on one implementation may actually signal an error on another, even if the argument is the same.

2.3 No exact fixnum-only arithmetic

R^{5}RS specifies in Section 6.2.2:

> If two implementations produce exact results for a computation that did not involve inexact intermediate results, the two ultimate results will be mathematically equivalent.

This makes it hard for a Scheme system to support only limited-precision integers, as it requires the system to mark results of overflows as inexact—which in turn usually means some loss of efficiency (if boxing is involved in the construction of inexact numbers or a conversion to a different representation) or additional loss of precision (if an additional bit in the fixed-size representation denotes inexactness).[1]

2.4 Numerical promotion loses information

The idea of a "numerical tower" suggests that the more general number types contain the more specific ones. In particular, there is usually a mechanism converting number types automatically when required for an operation, a process often called "numerical promotion", [15, Section 6.5.2.2, §§6, 7], [9, Section 5.6]. In Scheme, automatic conversion can occur both along the integer–number axis and along the exact–inexact axis. Also, information may get lost during any conversion of numbers, even *up* the tower. In MzScheme, for example, the following occurs:

```
(+ (expt 10 309) 0.0) ⟹ +inf.0
```

even though `(expt 10 309)` has an exact representation as an integer.

There are several reasons for loss of information. Bignums can be arbitrarily large, whereas fixed-precision floating-point formats are limited in range. Exact rationals can have arbitrary precision, whereas binary floating-point formats, even if they have arbitrary precision, can only represent binary fractions where the denominator is a power of two.

In effect, the intuition of the numerical tower as a chain of subsets is invalid for all actual Scheme systems having a floating-point representation or imposing limits on the number types.

2.5 Lack of "inexact control-flow"

R^{5}RS defines in Section 6.2.2 which numbers are exact:

> A number is exact if it was written as an exact constant or was derived from exact numbers using only exact operations.

Unfortunately, the following function is perfectly capable of returning an exact result given inexact input:

```
(lambda (x y) (if (< x y) -1 1))
```

Clearly, the reason is that the comparison < returns an exact result, either `#t` or `#f`. This is especially pernicious given that comparisons are one place where the inaccuracies of floating-point numbers may really hurt. In effect, the accuracy of the function's value is no greater than the accuracy of the input—but in Scheme's type system the result is treated as entirely exact.

[1] This seems to be why Bigloo 2.5c, for instance, has `(expt 2 50)` ⇒ `0` but `(exact? (expt 2 50))` ⇒ `#t` (in violation of the standard). Chicken 0/1082, which also does not support bignums, has `(expt 2 50)` ⇒ `1125899906842624.` and `(exact? (expt 2 50))` ⇒ `#f`.

To ensure that an exact result is not dependent on inexact operations the programmer either has to do a careful analysis of the program (in which case any run-time checking is irrelevant) or use exact comparison operations like the following:

```
(define (exact< x y)
  (if (and (exact? x) (exact? y))
      (< x y)
      (error ...)))
```

2.6 Exactness of numerical literals

Section 6.2.4 of R^{5}RS states: "If the written representation of a number has no exactness prefix, the constant may be either inexact or exact. It is inexact if it contains a decimal point, an exponent, or a "`#`" character in the place of a digit, otherwise it is exact."

The consequence is often that the global behavior of a program is governed by the presence or absence of a single decimal point: A program can become intolerably inaccurate through the presence of a decimal point, or intolerably slow through the omission of one. The example described in Figure 1 illustrates why this may be unfortunate.

2.7 Meaning of standard procedures

Some of the standard procedures defined in R^{5}RS only make sense for certain types of numbers, e.g., `gcd` for exact integers or `log` for inexact real or complex numbers.

This is a temptation for implementations to fill in the gap and define things like `(gcd 2.0 6.0)` in the "obvious" way, violating the intended meaning of standard procedures. In the following example (again run under MzScheme), the "greatest common divisor" might be greater than expected:

```
(gcd (expt 2 40) (expt 3 40))  ⟹ 1
(gcd (expt 2 40) (expt 3 40.)) ⟹ 2048.0
```

2.8 Exchanging numbers between Scheme systems

There is no guarantee that two R^{5}RS-compliant Scheme systems can successfully exchange numerical data via the written representations provided by the standard procedures. For exact integers, there seems to be no problem—provided the receiving system covers the range required by the sender. The notation `1/2` already poses a problem because rationals are not mandatory. The specification of `number->string` and `string->number` laudably caters to read/write invariance, but does so only for numbers written and read by the same Scheme system.

3 Exact arithmetic

The analysis in the previous section suggests that the numerical tower of R^{5}RS is not a good model for numerical computations in a computer program—at least not for all of them. Moreover, attributing exactness to numbers in the way R^{5}RS leads to inconsistencies.

In this section and the following two, we examine the consequences of splitting *operations* along the exact/inexact axis instead of the *numbers*. The exact arithmetic operations satisfy strong algebraic properties such as associativity, commutativity, distributivity, total

A typical example of surprises with mixed exact/inexact computation appeared in Sperber's Introductory Computing class. Students had to write a procedure for visualizing the Mandelbrot set. The task boils down to iterating the function $z \mapsto z^2 - c$ for different complex parameters c. For visualization, the procedure `draw-mandelbrot` enumerates points in a rectangle defined by upper left corner, width and height.

Many students observed that their program seemed to "hang" for some inputs, but not for others. This occurred when only literals without decimal point were used as operands for `draw-mandelbrot`—in which case the program computes the iteration using exact fractions. As the iteration progresses, the internal representation of the fraction gets very large very quickly.

Putting a decimal point into one of the numerical literals or placing an `exact->inexact` at almost any point in the program would fix things; there is no recognizably "right" place for it. Students find especially confusing that the seemingly "simpler"—integral—numblers cause problems, while the "more complicated" floating-point numbers do not.

The example illustrates the limited predictability of Scheme programs mixing exact and inexact numbers.

Figure 1. A real-world example

ordering etc. Initially, we consider the exact world only. We show how to add inexact operations later.

We take the following abstract numerical tower as the basis for our numerical operations:

$$\mathbb{Q} \supseteq \mathbb{Q}_{10} \supseteq \mathbb{Q}_2 \supseteq \mathbb{Z} \supseteq \mathbb{Z}_{\geq 0} \supseteq \mathbb{Z}_{>0}.$$

In this chain $\mathbb{Q}$ denotes the rational numbers, $\mathbb{Q}_b$ denotes the b-ary fractions, i.e. the set of rational numbers with denominator a power of b (binary fractions for $b = 2$ and decimal fractions for $b = 10$.)[2] $\mathbb{Z}$ denotes the integers, $\mathbb{Z}_{\geq 0}$ denotes the non-negative integers, and $\mathbb{Z}_{>0}$ denotes the positive integers.

While this view of the rational numbers may appear arbitrary or theoretical at first glance, it identifies and names the kinds of numbers that computer programs typically distinguish. In particular, positive and non-negative integers are so frequent in any sort of program that we propose to name them in the core language itself.

To relate the tower elements to machine representations, we use the following terminology, borrowed from R^{5}RS: *Fixnums* are the fixed-width machine representation for integers—denoted by fixnum. *Bignums* are the arbitrary-width exact representations for arbitrary integers, named bignum. *Flonums* are the fixed-precision floating-point machine representation for rational numbers, named flonum. Finally, *fractions* are the tuple representations for rational numbers, using bignum numerator and denominator, called fraction.

The relationships between the tower elements and the machine representations are as follows:

1. The fixnum representation implements a subset of $\mathbb{Z}$.
2. The bignum representation implements $\mathbb{Z}$, only limited by available memory.
3. The flonum representation implements a subset of $\mathbb{Q}_2$, possibly augmented by special objects like -0, $\pm\infty$ and NaN, which are not elements of $\mathbb{Q}$.
4. Human-readable representations are typically decimal fractions—elements of $\mathbb{Q}_{10}$—at least conceptually.
5. The fraction representation implements $\mathbb{Q}$, only limited by available memory.

We propose that the default operations on rational numbers, that means the standard procedures +, -, *, /, <=, etc., are all exact: Conceptually, they accept rational arguments and return rational results. Of course, implementations may take advantage of more efficient machine representations (employing fixnums and flonums) if possible, but conversion may only take place if no loss of information occurs in the process. Thus, the particular machine representation of a number is purely an efficiency issue.

In practice, this means the following:

- Each number object represents a unique, precisely defined rational number. Rational numbers have conceptually infinite precision.
- Different machine representations of the same rational number may coexist, but they are all equivalent. (Processing time may differ, of course.)
- Rational numbers are treated exactly the same way as R^{5}RS currently treats exact integers and rationals.
- The exact operations satisfy *all* algebraic properties (associativity, commutativity, distributivity, total ordering, etc.) of their mathematical counterparts.

4 Adding inexact arithmetic

Exact operations alone, even combined with explicit rounding, are not sufficiently efficient for many numerical computations. Therefore, the language should provide access to the underlying floating-point hardware, if available, through a default set of inexact operations. For example, `float+` would accept two flonums and return a flonum. By nature, `float+` sacrifices algebraic properties to gain efficient execution. However, by distinguishing exact and inexact operations explicitly, the actual arithmetic used becomes a property of the program, rather than a property of the numbers it processes. (Note that the arithmetic model remains a dynamic property in any language with exact and inexact numbers, even if operations are required to accept only all exact or all inexact argument values.)

By distinguishing exact and inexact operations explictly, we give up a potential source of code reuse: Even if an algorithm works for both exact and inexact operations alike, our proposal requires two different programs—one calling the exact operations, one calling the rounding operations. We are proposing to pay this price because sensible algebraic and numerical algorithms seem to be distinct most of the time.

Of course, practical implementations of the inexact operations will use a limited-precision floating-point representation for numbers. This raises the question of how these representations relate to the other representations for rational numbers. Do the floating-point representations form a subset of the rational representations? What about $\pm\infty$, NaN, and distinct -0? The issue of the special floating-point objects is central to this issue. We discuss $\pm\infty$ and NaN separately from distinct -0:

[2] What we denote as $\mathbb{Q}_b$ is not identical to the algebraic concept of "field of p-adic numbers."

4.1 The case against rational $\pm\infty$ and NaN

The special objects $+\infty$ and $-\infty$ are used in the floating-point world as a mechanism to carry on with a computation in the presence of overflow. They are usually the results of $\mathit{positive}/\mathit{tiny} = +\infty$ and $\mathit{positive}/(-\mathit{tiny}) = -\infty$, which can happen without the programmer being aware of it.

In the exact world, however, the only way of obtaining infinity is a division by zero. The question is whether the system should then signal an error, or return a special object representing infinity. An argument in favor of $\pm\infty$ is that they provide neutral elements for the minimum and maximum, i.e., `(min)` $\Rightarrow +\infty$, `(max)` $\Rightarrow -\infty$.

Nevertheless, an exact division by zero is virtually always a symptom of a genuine programming error or of illegal input data, and the introduction of infinity will only mask this error.

NaN ("not a number") is the strongest form of delaying an error message. NaN is a special object indicating that the result of an arithmetic operation is undefined; one way it could emerge is $(+\infty)+(-\infty) = \text{NaN}$. The advantage of returning NaN instead of raising an error is that the computation still continues, postponing the interpretation of the results to a more convenient point in the program. In this way, NaN is quite useful in numerical computations.

The problem with NaN is that the program control structure will mostly not recognize the NaN case explicitly. Assume we define comparisons with NaN always to result in `#f`, as IEEE 754 does, then

```
(do ((x NaN (+ x 1))) ((> x 10)))
```

will hang but

```
(do ((x NaN (+ x 1))) ((not (<= x 10))))
```

will stop, which is counter-intuitive and may be surprising.

While $\pm\infty$ and NaN are quite useful for inexact computations, there is a high price to pay when they are carried over into the exact world: The rational numbers must be extended by the special objects, and the usual algebraic laws will not hold for the extension anymore. Moreover, the special objects obscure exact programs by masking mistakes.

4.2 The case against rational -0

The purpose of distinguishing a "positive zero" $(+0)$ and a "negative zero" (-0) in a floating-point format is to retain the sign of numbers in the presence of underflow, e.g., $-0 = \mathit{positive}/(-\mathit{huge})$. Since comparisons must allow for tolerances, there is no real harm done identifying $+0$ (positive) with *the* zero, which is neither positive nor negative. The use of signed zeros simplifies dealing with branch cuts [11] and generally helps obtaining meaningful numerical output.

In the exact world, on the other hand, there is no underflow—only memory overflow. Even worse, adding one (or even two) signed "zeros" to the rational numbers completely destroys the rich, clean and simple algebraic structure which the rational numbers do posses. We briefly detail this mathematical fact.

The set $\mathbb{Q}$ of rational numbers equipped with the addition operation $+$ form an abelian group. This means the following:

(C) For all $x, y \in \mathbb{Q} : x+y = y+x$.

(A) For all $x, y, z \in \mathbb{Q} : (x+y)+z = x+(y+z)$.

(Z) There is $Z \in \mathbb{Q}$ such that for all $x \in \mathbb{Q} : Z+x = x$.

(I) For all $x \in \mathbb{Q}$ there is a $y \in \mathbb{Q} : x+y = Z$.

Now take elements $Z, Z' \in \mathbb{Q}$ such that $Z'+x = x$ for all $x \in \mathbb{Q}$ and also $Z+x = x$ for all $x \in \mathbb{Q}$. Then $Z = Z'+Z = Z+Z' = Z'$, where the second equation holds by (C). Consequently, there is *only one* element $Z \in \mathbb{Q}$ having property (Z). Therefore, this element receives the special name 0 (read "zero"). Now if we augment the set $\mathbb{Q}$ into $\mathbb{Q}'$ by forcibly adding another algebraic zero as in $\mathbb{Q}' = \mathbb{Q} \cup \{?\}$ where $?+x = x$ for all $x \in \mathbb{Q}'$ and $? \notin \mathbb{Q}$, then either property (C), or property (Z), or both get lost. This implies that property (I) at least suffers, because the uniqueness of y (which is in fact $-x$) gets lost. This carries on like wildfire, usually destroying nearly *all* algebraic properties at the same time; associativity may survive.

More generally, four different alternatives for dealing with '-0' in the exact world can be identified:

(a) Augment the rational numbers by one (or two) objects behaving like "a zero." Algorithmically, this means that all exact operations must dispatch on these special objects and define some action.

(b) Identify both floating-point values $+0$ and -0 with the rational number 0. In other words, exact operations treat ± 0 and 0 identically.

(c) Represent the floating-point value -0 by some negative rational number, say $-Z$. Conceptually, exact operations first replace -0 by the rational number $-Z$ and then do their work.

(d) It is an error to apply an exact operation to -0.

As explained above, the semantic cost of adding one or more "zeros" is quite high. This is a strong argument against alternative (a). In the other extreme, alternative (d) breaks the symmetry between positive and negative numbers. The problem with alternative (c) is to find a sensible definition of the rational equivalent of -0 (read "negative underflow.") A first approach might be: "-0 behaves like the smallest negative rational larger than any representable float." Unfortunately, there is no such rational number: Let $-f$ denote the largest representable negative float. Then $-f+1/n$, $n \in \{1, 2, 3, \ldots\}$, are not representable and increasingly close to $-f$. So there must be a gap between $-f$ and whichever rational number $-Z$ is choosen as the rational interpretation of '-0'—*unless* the definition reads: "Any $-Z$ for $-f < -Z < 0$ may be chosen as the rational interpretation of '-0';" an approach we do not pursue.

Whatever the choice, a negative number equivalent $-Z$ of -0 will behave surprisingly different from the float -0. For example, repeatedly squaring $-Z$ will soon exhaust memory and printing the square of $-Z$ will print unrecognizably, unless one is willing to sacrifice Scheme's facility to print rationals without loss of information.

Since alternatives (a), (c) and (d) are unattractive, alternative (b) appears to us as the least disadvantagous; there simply seems to be no place for $-0 \neq 0$ in the exact world of rational numbers.

5 Relating exact and inexact arithmetic

As the previous discussion has shown, the special floating-point values -0, $\pm\infty$, and NaN have no place in the exact world—they are not rational numbers. Hence, in the following, we assume that it is an error to apply an exact operation such as + to $\pm\infty$ or NaN, whereas ± 0 are both treated as 0 by the exact operations.

At this point, it is natural to ask whether the inexact numerical operations such as `float+`, `float-` etc. should accept all rational numbers, or only those represented as flonum. If the inexact operations only accept flonum arguments, a Scheme system must provide at least a conversion operation `rational->float`. Similarly, should the exact operations accept flonums (unless they are special values)? In other words, should the domains for exact and inexact operations be completely disjoint, with explicit conversion at all times? Three basic alternative kinds of "type permeability" seem to exist in this spectrum:

#1 The flonum representation is just another partial machine representation for rational numbers (plus special values), and all numerical operations, exact or inexact, accept all rational numbers as arguments. It is, however, an error to apply exact operations to $\pm\infty$ and NaN.

#2 As in #1, flonum is just another partial representation of rational numbers (plus special values), but inexact operations are *only* defined on flonum. Programs make use of the (rounding) operation `rational->float` to convert explicitly.

#3 The flonum representation is completely distinct from implementation of rationals. In other words, the exact operations are not defined on flonum and the inexact operations are undefined for the non-flonum rational numbers. Programs use of `float->rational` and `rational->float` to convert explicitly.

All three alternatives could support a `float?` predicate that answers `#t` for all flonum arguments—including $\pm\infty$ and NaN. A `rational?` predicate would probably behave differently in the different alternatives: Whereas it would answer `#t` to all numbers except for $\pm\infty$ and NaN in #1 and #2, it would naturally be a converse of `float?` in #3. Probably, a `float-not-rational?` predicate that identifies $\pm\infty$ and NaN would also be useful.

Alternatives #2 and #3 both also require distinct *external* representations for flonum and non-flonum rationals. If an external representation denotes a flonum, it may also be desirable to require representation information to accurately determine the meaning of the literal. (More on the issue of external representation in Section 6.6.) Alternatively, all numerical literals denote rational numbers, and the program must convert them to flonum representation explicitly via `rational->float`.

Alternatives #1 and #2 can both be implemented as conservative extensions of R^{5}RS by the following measures:

- Support integers and rationals of arbitrary precision.
- Have all R^{5}RS numerical operations convert flonum arguments to fraction before proceeding. (Or assert correctness by other means.)
- Interpret "inexact" as "float." Specifically, take `inexact?` to mean `float?` and `exact?` as ¬`inexact?`. Define `exact->inexact` and `inexact->exact` as follows:

```
(define (exact->inexact n)
  (if (float? n)
      n
      (rational->float n)))

(define (inexact->exact n)
  (cond
   ((float? n) (float->rational n))
   ((number? n) n)
   (else
     (error ...))))
```

  Note that `(number?` NaN`)` ⇒ `#f` and `(number?` $\pm\infty$`)` ⇒ `#f`, while `(float?` NaN`)` ⇒ `#t` and `(float?` $\pm\infty$`)` ⇒ `#t`.

- Finally, add operations on flonum with a `float` prefix.

(Of course, `inexact?`, `exact?`, `exact->inexact`, and `inexact->exact` serve no purpose in this new organization of numbers and should disappear eventually.)

The only problem is that of literals: Alternative #1 would work most intuitively if unannotated numerical literals would always represent their rational counterparts exactly. Unfortunately, R^{5}RS requires that the presence of a decimal point or an exponent forces a literal to denote an inexact, and, thus, a floating-point number. Therefore, a true conservative extension still requires that "exact" numerical literals carry a `#e` prefix.

In any case, all alternatives feature full reproducibility for exact computations, and much-improved transparency because the program source code clearly shows when floating-point arithmetic happens. (As for the example in Figure 1: In our design, the program would *always* compute slowly. However, the program now behaves in a much more consistent and less confusing manner, and the cause for the problems is much easier to explain than with R^{5}RS, as is the remedy.)

6 Useful numerical operations

In this section, we disucss alternatives to R^{5}RS's default set of numerical representations. Any such design necessarily represents a subjective choice, however. It should be rich enough to be convenient (e.g. having both < and >) but leave less frequently used operations (like `gcd` and `lcm`) to specialized libraries.[3] Here is possible list of exact operations to be present in the core language of a Scheme system:

`rational?` `decimal-fraction?` `binary-fraction?`
`integer?` `non-negative-integer?` `positive-integer?`
(Section 6.1)
`negative?` `zero?` `non-negative?` `positive?`
`compare` [$= \operatorname{sign}(x-y)$] (Section 6.2)
`<` `<=` `=` `>=` `>` `min` `max` `sign` `abs`
(`if-sign` *x negative zero positive*) (Section 6.2)
`+` `-` `*` `/` `^` [alias `expt`]
`floor` `ceiling` `truncate` `extend` `round` (Section 6.3)
`round-fraction` `floor-log-abs` (Section 6.4)
`div` `mod` (Section 6.5)
`numerator` `denominator`
`string->rational` `rational->string` (Section 6.6)

[3] Of course, fractional arithmetics requires a gcd operation internally—but including rarely used operations in the default set carries a conceptual cost.

We discuss the major deviations from R^{5}RS.

6.1 Numbers

The type predicates `rational?` `decimal-fraction?` `binary-fraction?` `integer?` `non-negative-integer?` `positive-integer?` reflect the abstract chain of numbers as introduced in Section 3.

As mentioned already in Section 3, non-negative and positive integers are exposed because of their ubiquitous nature. Concerning decimal and binary fractions, refer to Section 6.4.

6.2 Comparisons

The additional comparison operations increase programming convenience. With respect to R^{5}RS, there are two major additions: The `compare` procedure and the `if-sign` special form dispatching on the sign of a rational number.

`Compare` has been included for efficiency. All other comparisons can be expressed in terms of a single call to `compare`, which can be implemented without allocating any intermediate objects at all.

`If-sign` has been included because a frequent task in programming is distinguishing between the three possible results of a comparison.

6.3 Rounding rationals to integers

For rounding rationals into integers, the procedures `floor`, `ceiling`, `truncate`, `extend` and `round` provide the rounding modes towards $-\infty$, $+\infty$, 0, $\pm\infty$ and towards the nearest integer. The precise mathematical definitions of these functions are the obvious ones, with the exception of breaking ties in `round`, which breaks ties towards even, just like R^{5}RS and IEEE 754.

All of these operations are useful and common in numerical programs: Breaking ties towards even and towards zero is symmetric in the sense that $\rho(-x) = -\rho(x)$ for all x, where ρ denotes the rounding function. Breaking ties towards $-\infty$ appears naturally in `div` and `mod` as defined in Section 6.5. Finally observe that rounding with breaking ties towards $\pm\infty$ is naturally related to floor and ceiling by $\lceil x-1/2\rceil$ and $\lfloor x+1/2\rfloor$.

6.4 Binary and decimal fractions

By providing a convenient function for rounding rationals into binary and decimal fractions, programs can easily implement floating-point operations of arbitrary precision in the absence of, or in addition to, proper floats. Among others, this provides a natural way of defining external representations for binary and decimal fractions accurately and portably. (A proposal is in Section 6.6.) We propose that

`(round-fraction` *base mantissa round x*`)`

maps the rational x into a number that has *mantissa* significant digits in its *base*-ary expansion and where rounding has been performed by applying the procedure *round* mapping rationals into integers. Figure 4 shows a possible implementation in R^{5}RS, assuming the presence of (bignum) rational arithmetics.

More explicitly, `(round-fraction` $b\ m\ \rho\ x$`)` should result either in 0, or in a number of the form

$$\hat{x} = \mathrm{sign}(x) \cdot (\hat{x}_0.\hat{x}_1 \cdots \hat{x}_{m-1})_b \cdot b^{\hat{e}}, \tag{1}$$

for b-ary digits $\hat{x}_0, \ldots, \hat{x}_{m-1} \in \{0, \ldots, b-1\}$, $\hat{x}_0 \neq 0$, and integer $\hat{e}$. Clearly, this only makes sense for integer b and m where $b \geq 2$ and $m \geq 1$.

Now consider the case $\hat{x} \neq 0$. Then

$$1 = (1.0\cdots)_b \leq (\hat{x}_0.\hat{x}_1 \cdots \hat{x}_{m-1})_b < b = (10.0\cdots)_b.$$

This implies $0 \leq \log_b(\hat{x}_0.\hat{x}_1 \cdots \hat{x}_{m-1})_b < 1$, from which follows

$$\lfloor \log_b |\hat{x}| \rfloor = \hat{e}.$$

This is the primary reason for proposing that

`(floor-log-abs` $b\ x$`)`

computes the largest integer e such that $b^e \leq |x|$ for integer b, $b \geq 2$, and non-zero rational x. Note that e is negative if and only if $|x| < 1$.

Coming back to `round-fraction`, define for $x \neq 0$

$$\hat{x} = \rho\left(x b^{m-e-1}\right) \cdot b^{-(m-e-1)}, \quad e = \lfloor \log_b |x| \rfloor. \tag{2}$$

Clearly, this definition can only result in the form (1) if the rounding function $\rho(-)$ is well-behaved. For this reason, we require that $\rho(u)$ is integer and $|\rho(u) - u| < 1$ for all rational u. This is the case for `round`, `floor`, `ceiling`, `truncate`, and `extend`. (In the case of `round`, even the tighter bound $|\rho(u) - u| \leq 1/2$ holds.) For $x = 0$ define $\hat{x} = 0$.

Under these conditions, the following error bound holds:

$$|\hat{x} - x| < b^{-m+1}|x|.$$

Proof:

$$\begin{aligned} |\hat{x} - x| &= |\rho\left(x b^{m-e-1}\right) b^{-(m-e-1)} - x| \\ &= b^{-(m-e-1)} |\rho\left(x b^{m-e-1}\right) - x b^{m-e-1}| \\ &< b^{-(m-e-1)} \\ &\leq b^{-m+1}|x|. \end{aligned}$$

It remains to be shown that the conditions on $\rho(-)$ imply the form (1). Observe that a positive u is never rounded into a negative $\rho(u)$, and vice versa. This means that we only need to consider $x > 0$. In this case, $b^e \leq x < b^{e+1}$ by definition of e, which implies $b^{m-1} \leq x b^{m-e-1} < b^m$. Applying ρ, we obtain

$$b^{m-1} \leq \rho(x b^{m-e-1}) \leq b^m,$$

because $\lfloor u \rfloor \leq \rho(u) \leq \lceil u \rceil$. Hence, we have shown that $\hat{x}$ has at most m non-zero digits in its b-ary expansion.

Note that `round-fraction` ignores several details of actual floating-point formats: The exponent of `round-fraction` is unlimited in magnitude, which means overflow and mantissa denormalization ($\hat{x}_0 = 0$) do not occur. Also underflow, the production of a number of magnitude too small to be represented, is not detected; it is simply rounded to zero.

6.5 `Div` and `mod`

Given an unlimited integer type, it is a trivial matter to derive signed and unsigned integer types of finite range from it by modular reduction. For example, arithmetic using 32-bit signed two's-complement behaves like computing with the residue classes "mod 2^{32}," where the set $\{-2^{31},\ldots,2^{31}-1\}$ represents the residue classes. Likewise, unsigned 32-bit arithmetic also behaves like computing "mod 2^{32}," but using a different set of representatives: $\{0,\ldots,2^{32}-1\}$.

Unfortunately, the R^{5}RS-operations `quotient`, `remainder`, and `modulo` are not ideal for this purpose. In the following example, `remainder` fails to transport the additive group structure of the integers over to the residues modulo 3.

```
(define (r x) (remainder x 3))
(r (+    -2     3 ))  ⟹  1
(r (+ (r -2) (r 3)))  ⟹ -2
```

In fact, `modulo` should have been used, producing residues in $\{0,1,2\}$. For modular reduction with symmetric residues, i.e. in $\{-1,0,1\}$ in the example, it is necessary to define a more complicated reduction altogether.

Therefore we propose operations div and mod (with Scheme counterparts `div` and `mod`), defined on all integers x,y, by the following properties

$$x = (x \operatorname{div} y)\cdot y + (x \bmod y), \tag{3}$$

$$\begin{array}{rcccll} 0 & \leq & (x \bmod y) & < & y & \text{if } y>0, \\ y/2 & \leq & (x \bmod y) & < & -y/2 & \text{if } y<0, \end{array} \tag{4}$$

$$x \operatorname{div} y \text{ is integer, and } x \operatorname{div} 0 = 0. \tag{5}$$

In other words, the sign of the modulus y determines which system of representatives of the residue class ring $\mathbb{Z}/y\mathbb{Z}$ is being chosen, either non-negative ($y>0$), symmetric around zero ($y<0$), or the integers ($y=0$).

The definition above implies

$$x \operatorname{div} y = \begin{cases} \lfloor \frac{x}{y} \rfloor & \text{if} \quad y>0, \\ 0 & \text{if} \quad y=0, \\ \lceil \frac{x}{y} - \frac{1}{2} \rceil & \text{if} \quad y<0. \end{cases}$$

This simplicity is the reason why the definition can be extended literally to define div and mod for all rational x,y. Mathematically, it even makes sense for all real x,y. For example, $(x \bmod 2\pi)$ and $(x \bmod -2\pi)$ both reduces x modulo 2π, and

$$0 \leq (x \bmod 2\pi) < 2\pi \text{ and } -\pi \leq (x \bmod -2\pi) < \pi.$$

Since div and mod offer both conventions which make sense, the R^{5}RS procedures `modulo`, `remainder`, and `quotient` can easily be defined in terms of `div` and `mod`. Of course it is also possible the other way around, albeit with more effort. Figures 2 and 3 show the definitions, respectively.

```
(define (quotient  n1 n2)
  (* (sign n1) (sign n2) (div (abs n1) (abs n2))))

(define (remainder n1 n2)
  (* (sign n1) (mod (abs n1) (abs n2))))

(define (modulo n1 n2)
  (* (sign n2) (mod (* (sign n2) n1) (abs n2))))
```

Figure 2. Defining `quotient`, `remainder`, `modulo` in terms of `div`, `mod`, `sign`, and `abs`.

```
(define (div x y)
  (cond
   ((positive? y)
    (let ((n (* (numerator x)
                (denominator y)))
          (d (* (denominator x)
                (numerator y))))
      (if (negative? n)
          (- (quotient (- d n 1) d))
          (quotient n d))))
   ((zero? y)
    0)
   ((negative? y)
    (let ((n (* -2
                (numerator x)
                (denominator y)))
          (d (* (denominator x)
                (- (numerator y)))))
      (if (< n d)
          (- (quotient (- d n) (* 2 d)))
          (quotient (+ n d -1) (* 2 d)))))))

(define (mod x y)
  (- x (* (div x y) y)))
```

Figure 3. Defining `mod` and `div` using R^{5}RS, assuming exact rational arithmetics.

6.6 External Representations

We discuss some of the issues regarding external representatives arising from our design proposal in this section.

External representations occur in several contexts:

- literals in program source code,
- the output of `write` and the input of `read`,
- numbers printed out for human readers,
- numbers printed for consumption by other (non-Scheme) programs and read from other programs.

Since the number formats used for consumption by humans and non-Scheme programs vary wildly and uncontrollably, they are properly the subject of one or probably several libraries and beyond the scope of this paper. We focus on literal syntax and on the syntax used by `read` and `write`.

In Scheme, to preserve some of the desirable properties of the language, the literal syntax must be compatible with the format used by `read` and `write`.

The simple-minded approach to the external-representation issue is to just have one uniform external representation for all machine number formats—each representation stands for a unique rational number, and converting a number to its representation is an exact operation. However, many floating-point numbers have quite long representations as fractions, making this choice prohibitive in terms of both space (for storage of the representation) and time (for converting back and forth between the numbers and their representation).

```
(define (floor-log-abs base x)
  (define (log b x e b^e offset)
    (let ((b^e+1 (* b^e b)))
      (if (> b^e+1 x)
          (if (= b^e x) e (+ e offset))
          (log b x (+ e 1) b^e+1 offset))))
  (let ((abs-x (abs x)))
    (if (>= abs-x 1)
        (log base abs-x 0 1 0)
        (- (log base (/ 1 abs-x) 0 1 1)))))

(define (round-fraction base mantissa round x)
  (if (zero? x)
      0
      (let ((k (- mantissa
                  (floor-log-abs base x)
                  1)))
        (* (round (* x (expt base k)))
           (expt base (- k))))))
```

Figure 4. `Floor-log-abs` and `round-fraction` as defined in Sections 6.3 and 6.4, implemented in R^{5}RS, assuming rational arithmetics.

Hence, it is desirable to be able to use a shorter, floating-point (in the true sense of "using a point") external representation for numbers, preferably using the familiar decimal-point format. In that case, read/write invariance requires tagging the result explicitly as a floating-point number. Moreover, to better support the exchange of external representations between different Scheme systems, or to support distinguishing between several machine floating-point formats used by a single Scheme system, it is desirable to provide information about the nature of the floating-point format used.

We suggest using a suffix indicating the length of the binary mantissa of the floating-point format. Thus, In our proposal, `0.7` would always denote 7/10 (unless R^{5}RS compatibility is important, see Section 5), whereas the IEEE 754 64-bit float closest to 0.7 would print as `0.7|52`, which is equal to `3152519739159347/4503599627370496`. We call this format the *mantissa-width tagged format*.

From the point of view of communication, the mantissa-width tagged format is not so much an indicator for "floating point" but rather a source coding (compression) method for a frequently used subset of the rational numbers—binary fractions. The mantissa-width tagged format for binary fractions achieves accuracy without loss of performance.

The mantissa-width tagged format can be specified accurately in terms of the procedures `round-fraction` and `round` of Sections 6.4 and 6.3. To be specific, we propose procedures `string->rational` and `rational->string` (serving the function of R^{5}RS's `string->number` and `number->string`) that convert between internal and external representations of rational numbers. Apart from the usual formats (base 2/8/10/16, fractions via `/`, and decimal scientific "e"-notation), `string->rational` understands the number syntax

scientific`|`*mantissa*

and interprets it as

`(round-fraction 2` *mantissa* `round` *scientific*`)`

`Rational->string` and `string->rational` satisfy the "read/write-invariance" property of R^{5}RS: For each rational number x (in the sense of `rational?`), the following holds:

`(= (string->rational (rational->string` x`)) `x`)`

(Note that our `=` is an exact comparison, unlike the `=` of R^{5}RS, which is the reason R^{5}RS formulates this property in terms of `eqv?`.)

To summarize, we suggest the following (partly departing from R^{5}RS):

- Each external number representation without annotation denotes exactly the rational number the "learned in high school interpretation" would assign it. That is, `0.7` $= 7/10$ and `1.3e-2` $= 13/1000$.
- The mantissa-width tagged format specifies a *binary* fraction (like a floating point number) by *decimal* digits: `0.7|5` $= 11/16$ and `0.7|52` $= 3152519739159347 \cdot 2^{-52}$.
- The `#e` and `#i` prefixes go away.

Note that we expect the mantissa-width tagged format to occur only rarely in numerical literals—the programmer can simply specify a rational number and rely on the automatic conversion for `float` operations.

The R^{5}RS requirement that `number->string` must use the minimum number of digits for decimal-point external representations must be adjusted for `rational->string`, as there might be several different representations for the same number. For example, `11/32` $=$ `0.34|4` $=$ `0.34375`: Although the mantissa-width tagged format is shorter, the purely decimal format is arguably clearer.

Consequently, we propose to require the minimum number of digits only within one particular number format, but give the implementations the freedom to choose the format. Nevertheless, printing with the absolute minimum of characters is also possible and even computationally inexpensive.

7 Implementation Issues

In this section, we address the most important implementation issues that arise with our proposal:

7.1 Exact operations on flonums

In design alternatives #1 and #2, numbers represented as flonums will be converted into fractions when an exact operation requires it. This might lead to surprises in terms of time and memory consumed, because exact representations can and generally do grow quickly with arithmetic depth. This is the price of exactness.

However, if problems arise from exact operations on flonums, they are easy to detect (slow execution) and have a specific remedy: Replace exact operations by inexact operations and investigate numerical stability. R^{5}RS, on the other hand, makes it much harder to identify and systematically fix this kind of problems because exactness is not a static property of the program. In other words, the programmer must investigate the run-time propagation of inexactness in order to understand the algorithm actually being executed.

7.2 Generic arithmetic

The exact arithmetic operations need to dispatch on the representations of their arguments—a typical implementation will at least use separate representations for fixnums, bignums, and true fractions. This is no different from the situation in R^{5}RS, and a Scheme system can employ the same technique as before to perform the dispatch—for example, via exhaustive case analysis or a suitable exception system.

7.3 Coercion of constants

If number literals containing a decimal point (and without a mantissa-width specification) are interpreted as rationals, and floating-point operations accept rational arguments (as in design alternative #1), the implementation will typically need to convert the rational number to a floating-point representation. This may be a relatively expensive operation, and a straightforward program may perform it often. To reduce the cost, an implementation could memoize the floating-point approximation of a rational number, or perform a static analysis to determine what literals are used exclusively as arguments to floating-point operations. We conjecture that a simple analysis would be quite effective for most realistic programs.

7.4 Fixnum arithmetic

Many Scheme implementations already use fixnum arithmetic to optimize common-case numerical operations. However, implementations might want to offer exclusively fixnum arithmetic to optimize away the generic-arithmetic dispatch and the overflow detection. Doing this in the default set of numerical operations on exact numbers is already in violation of R^{5}RS. (See Section 2.)

Thus, the best way of offering fixnum-only operations would be through a set of separate procedures, analogous to the floating-point operations, with their algebraic meaning defined as calculating "mod $\pm 2^w$", $w \in \{8, 16, 32, 64\}$, as proposed in Section 6.5.

7.5 Floating-point representation

We have said nothing about the particular machine floating-point representation a Scheme system may use or should be required to use by a standard. This is a touchy issue—requiring, say, a particular IEEE 754 representation would lead to completely reproducible computations, but, depending on the hardware a program runs on, results in an unacceptable loss in either accuracy or efficiency [4, 12] and might pose a considerable obstacle for implementations on platforms not supporting this representation natively.

For this reason, we would expect a standard to specify that the floating-point operations use the widest floating-point format the underlying hardware supports efficiently. In practice, this would probably mean IEEE 754 double extended on the Intel x87 or the 68xxx architecture, and IEEE 754 double on, say, the PowerPC, or the Alpha.

Of course, implementations could also offer sets of floating-point operations specific to a specific machine representation or with parameters (e.g. multiprecision.) However, as few programs seem to require this degree of control, it should probably not be included into the core language by default.

7.6 Floating-point storage

The choice of the storage format for large quantities of floating-point numbers is independent of the choice of the format used for computations. Uniform vectors that explicitly specify the floating-point format used, such as those proposed in SRFI 4 [8] are an appropriate mechanism for this.

7.7 Mantissa-width tagged format

Reading the mantissa-width tagged format proposed in Section 6.6 can be done efficiently using Clinger's method [3, 2].

Similarly, printing the mantissa-width tagged format using the minimum number of total digits can be reduced to Burger and Dybvig's efficient method for printing a binary fraction as an approximate decimal fraction [19, 1]. The most important difference is that the mantissa width may vary with the number being printed. In effect, the mantissa-width tagged format can often be shorter, as for example in `1e9|1` $= 2^{30}$. Whether the system should really use the mantissa-width tagged format in this case is a different matter.

8 Related Work

Some Scheme implementations targeted at high performance programs—such as Chez Scheme [6], and Bigloo [17]—offer specialized numerical operations for floating-point numbers. This underlines the need for separating floating-point arithmetic from the usual generic arithmetic for performance reason, but does not really address the concerns raised in this paper: The remaining numerical operations are unaffected in these systems. Gambit-C [7] offers a declaration which locally declares all R^{5}RS numerical operations to perform floating-point arithmetic—again, for performance reasons.

The teaching languages of DrScheme [5] use exact arithmetic by default, to spare beginning students the confusion of programming with mixed exact and inexact floating-point arithmetic.

Objective Caml [14] keeps the domains and types for floating-point numbers completely separate from that of integers: A program cannot use them interchangeably, it must explicit convert. The floating-point operations have names different from the integer operations. (`+.` for floating point addition, etc.) Keeping the floating-point numbers separate from the rest is easier in Objective Caml than it is in Scheme because Caml does not have built-in rational numbers. Hence, there is no choice but the read `0.7` as a float.

Haskell 98 [10] also has a sophisticated type hierarchy for its numerical types, including rational numbers and single- and double-precision floating-point numbers. It keeps the various numerical types separate, but uses its type class mechanism to use a single set of operators for all numerical types and make parts of the numerical domains look like subtype hierarchies. Just like our proposal, Haskell mandates that a literal containing a decimal dots represents its corresponding rational number accurately. Two methods `fromInteger` and `fromRational`, overloaded over their respective result types, negotiate between literals and the contexts that receive them. Ambiguities concerning the numerical types are frequent, which is why the `default` declaration can specify a strategy for resolving them.

Common Lisp [18] does not have inexactness as a property of numbers orthogonal to the representation type. However, numerical operations will always convert rational arguments to float arguments

if any other arguments are floats. Comparisons between floats and rationals always convert the floats to rationals. Unlike Scheme, Common Lisp does at least give a recommendation for the minimum precision offered by the various floating-point operations, which, we conjecture, reduces the variance between different Common Lisp systems considerably. However, the basic arithmetic operations are still overloaded and do not always respect the various algebraic laws.

Mathematica [20] provides an arbitrary-precision floating-point representation and applies a mechanism of decreasing precision during inexact computation. In practice, however, this approach suffers from the same weaknesses as R^{5}RS: When inexact numbers enter the computation, it is usually time to design a new program. Moreover, the automatic decreasing of precision makes it difficult to run entire computations at a higher precision; a stray `1.0` (default precision) instead of a `N[1, 50]` (high precision) propagates its low precision uncontrollably, usually ruining the calculation.

An alternative approach to preserve read/write invariance (and a number of the other issues raised in this paper) would be to *fix* the floating-point representation in the language specification once and for all, as for example has been done in Java [9]. In that case, no tagging is necessary. The controversy around this approach suggests against it [12].

Scheme has long been one of the few languages to specify that a round-trip of conversion of a number to an external representation and back should preserve that number. Hence, it comes as little surprise that the most important publications about efficient and accurate algorithms to achieve this purpose come from the Scheme community [3, 2, 19, 1].

9 Conclusion

In Section 1.1, R^{5}RS says:

> Scheme's model of arithmetic is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. [...] Thus the distinction between integer and real arithmetic, so important to many programming languages, does not appear in Scheme.

We have shown that the behavior of realistic programs is in fact very much dependent on the particular number representations chosen by an implementation. The distinction between integer and real arithmetic is important to many other languages because it is important to programs. Following this design guideline, R^{5}RS makes is very difficult to write portable programs employing inexact arithmetic: Inexact arithmetic is too underspecified to allow a programmer to predict what a particular program will do running in different Scheme implementations. At the heart of the problem is the notion of inexact numbers itself—a more useful basis for the design of a set of numerical operations is attributing inexactness to the operations rather than the numbers.

We have designed the basis for such a set of numerical operations, and identified design alternatives within its framework. The most important property of our design is that the default numerical operations are always exact. Floating-point arithmetic is relegated to a separate set of operations. Most of the choices available within the design concern the degree of separation between the inexact and exact worlds. However, all of the alternatives we propose have more pleasant properties than what R^{5}RS currently requires—in particular, greater transparency and full reproducibility for exact computations. They also require similar, if not less implementation effort. We have also identified some weaknesses in the set of numerical operations offered by R^{5}RS, and proposed alternatives.

Arguably, the result is still "strange" in that it is unlike basically every other programming language. We conjecture that this difference is good and necessary: In particular, most programming languages do not offer infinite-precision integers and rational numbers at all, which reduces the design space, but comes with its own problems: Limited precision of the various numerical types along with implicit coercion rules often cause programming errors and non-reproducible behavior. Of the languages that do support infinite-precision integers and rationals, only Common Lisp stands out, which takes a less principled but otherwise similar approach to Scheme. We conjecture that programmers experience similar surprises in Common Lisp as in Scheme. However, given Common Lisp's much tighter specification and as much fewer Common Lisp systems exist than Scheme systems, these surprises may not matter as much in practice. All in all, we believe that Scheme is special enough to warrant a special design for its numerical operations.

10 References

[1] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116, Philadelphia, PA, USA, May 1996. ACM Press.

[2] William D. Clinger. How to read floating point numbers accurately. In PLDI 1990 [16], pages 92–101.

[3] William D. Clinger. How to read floating point numbers accurately. In Kathryn S. McKinley, editor, *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection.* ACM, April 2004. SIGPLAN Notices 39(4).

[4] Joseph Darcy. Adding IEEE 754 floating point support to Java. Master's thesis, University of California at Berkeley, 1998. `http://www.cs.berkeley.edu/~darcy/Borneo/spec.html`.

[5] *PLT DrScheme: Programming Environment Manual*, May 2004. Version 207.

[6] R. Kent Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998. `http://www.scheme.com/csug/index.html`.

[7] Marc Feeley. *Gambit-C, version 3.0, A portable implementation of Scheme*, 3.0 edition, May 1998. `http://www.iro.umontreal.ca/~gambit/doc/gambit-c.html`.

[8] Marc Feeley. SRFI 4: Homogeneous numeric vector datatypes. `http://srfi.schemers.org/srfi-14`, May 1999.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[10] Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition`, December 1998.

[11] William Kahan. Branch cuts for complex elementary functions, or much ado about nothing's sign bit. In A. Iserles and M.J.D. Powell, editors, *The State of the Art in Numerical Analysis*. Clarendon Press, 1987.

[12] William W. Kahan and Joseph D. Darcy. How Java's floating-point hurts everyone everywhere. http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf, March 1998.

[13] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[14] Xavier Leroy. *The Objective Caml system release 3.08 , Documentation and user's manual*. INRIA, France, July 2004. http://pauillac.inria.fr/caml.

[15] International Standards Organization. Programming language — C, 1999. ISO/IEC 9899.

[16] *Proc. Conference on Programming Language Design and Implementation '90*, White Plains, New York, USA, June 1990. ACM.

[17] Manuel Serrano. *Bigloo—A "practical Scheme compiler"—User manual for version 2.6d*, April 2004. http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html/.

[18] Guy Steele. *Common LISP: The Language*. Digital Press, Bedford, MA, 2nd edition, 1990.

[19] Guy L. Steele and Jon L. White. How to print floating-point numbers accurately. In PLDI 1990 [16], pages 112–126.

[20] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, 5th edition, 2003.

The R6RS Status Report

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Editors' note: This article is the lightly edited text of the progress report submitted by the Scheme Language Editors Committee to the Scheme Language Steering Committee on September 2, 2004. We have included it in the workshop proceedings to represent the concluding presentation of the workshop on the state of the standardisation effort by the editors committee.

The members of the Scheme Language Editors Committee are:

Marc Feeley, editor in chief (Université de Montréal)
Will Clinger (Northeastern University)
Kent Dybvig (Indiana University)
Matthew Flatt (University of Utah)
Richard Kelsey (Ember Corporation)
Manuel Serrano (INRIA)
Michael Sperber (DeinProgramm)

The members of the Scheme Language Steering Committee are:

Alan Bawden (Brandeis University)
Guy L. Steele Jr. (Sun Microsystems)
Mitch Wand (Northeastern University)

–Waddell & Shivers

At the 2003 Scheme workshop in November, the strategy committee (Alan Bawden, Will Clinger, Kent Dybvig, Matthew Flatt, Richard Kelsey, Manuel Serrano, Mike Sperber) was given a mandate to nominate a steering committee and an editors committee to work on the R6RS standard. In January 2004, the editors committee was nominated: Feeley (editor in chief), Clinger, Dybvig, Flatt, Kelsey, Serrano, and Sperber.

On January 19, a private mailing list was created to keep a record of the email exchanges between the editors. Although some editors suggested that a more open process would be desirable, we chose to keep this mailing list private to avoid outside interference and keep the process disciplined and focused. Sometime in the future the archive of the discussions will be made public so that the reasons for the design decisions are clear.

Because of the expected difficulty in managing productive discussions for a seven-member committee by email, we adopted some ground rules for ensuring progress. If an editor does not participate in an email discussion within a reasonable time limit (which was set to seven days), then the other editors may assume that editor does not have an opinion on the subject (or does not want to voice his opinion), and can be ignored (in the discussion, in a vote, *etc.*). I think this has been helpful to create a certain pressure to keep up-to-date in the discussion.

The subject of backward compatibility was also discussed early on. That is, will R5RS code work unchanged in an R6RS-compliant Scheme implementation? Our position is that backward compatibility is desirable but that there may be some incompatibilities (for example at the lexical syntax level) that prevent R5RS code from working under R6RS. Our first objective is to improve the Scheme language. Backward compatibility, while important, is a secondary objective.

We then set off on our first technical task: come up with a list of goals that is more precise than the one in the draft charter (which had four items: produce a core Scheme specification, define a module system, define a macro system, and designate library modules). Our plan was to use this list of goals (1) to organize the design process, and (2) to identify which changes were uncontroversial (and thus easier to standardize) and which would require considerable effort (and where consensus might not be achievable during the R6RS design process).

All the editors were polled to get a list of specific issues that they thought needed to be addressed in the R6RS design process (*i.e.*, features the committee should consider adding/removing). At the end of March, we had merged all the editors' lists into a single list with each editor's position. At this point, there had been very little technical discussion of these issues (on purpose), so that we could order the issues and discuss them in a disciplined way. As suggested by Will Clinger, the list was organized into the following categories:

- Deletions of R5RS-Scheme features;
- Incompatible changes to R5RS Scheme;
- Extensions that could be entirely compatible with R5RS Scheme

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

  - but would break some implementation-specific extensions;
  - but would be controversial and aren't worth it;
  - that are controversial or difficult but necessary;
  - that are probably uncontroversial.

Below is the list of issues, without each editor's position. Note that this list is still open to be expanded as new issues arise in the design process.

Deletions from R5RS

- remove `transcript-on` and `transcript-off`
- remove `force` and `delay`
- remove multiple values

Incompatible changes to R5RS

- make syntax case-sensitive

Extensions that would break implementation-specific features

- specify evaluation order
- support for processes
- support for network programming
- object-oriented programming
- external representation for records
- serialization

Extensions to R5RS (controversial and probably unnecessary)

- pattern matching / destructuring
- abstract data type for continuations
- composable continuations
- box types
- uninterned symbols
- extended symbol syntax
- add `letrec*`, define internal `define` in terms of it
- optional and keyword arguments as in DSSSL

Extensions to R5RS (controversial or difficult but necessary)

- module system
- non-hygienic macros
- records
- mechanism for new primitive types
- Unicode support
- errors and exceptions
- require a mode where "it is an error" means "an error is signaled"

Extensions to R5RS (probably not terribly controversial)

- multiline comments
- external representation for circular structures
- `#!eof`
- more escape characters
- require that `#f`, `#t`, and characters be followed by a delimiter
- `case-lambda`
- `cond-expand`
- allow the name of the macro being defined in `syntax-rules` to be arbitrary (or _)
- allow continuations created by `begin` to accept any number of values
- tighten up specification of `eq?` and `eqv?` (or otherwise address their portability problems)
- tighten up specification of inexact arithmetic
- add `+0`, `-0`, `+inf`, `-inf`, `+nan`
- bitwise operations on exact integers
- SRFI 4 homogeneous numeric vectors
- specify dynamic environment
- operations on files
- binary I/O or new I/O subsystem entirely
- string code
- regular expressions
- command-line parsing
- hash tables
- library for dates
- system operations

Editorial changes

- split language into core and libraries

Additional extensions

- expression comments
- subset of Common Lisp `format` (in a library)

Because of the central role of the module system and its probable use in splitting the Scheme language into a core and libraries, we decided that the most pressing issue was the design of the module system. Our starting point was the "strawman module system" proposed by Flatt, which is based on the MzScheme system. Various aspects of the proposed system were discussed, mainly to understand it better and to add constructive criticism. Because many aspects are interrelated, we did not achieve consensus on any specific aspect (nor did we really try to achieve it given that this is early in the design process).

Over May and June, the discussion on the module system was slow and only two of the seven editors were active. At the end of June, I suggested that the reason for this apathy might be a lack of practical experience with the proposed module system (the two editors that were active both had experience with the MzScheme module system). I proposed that we should work on building a portable implementation of the module system so that the editors can all experiment with it in our own Scheme implementations. This would get the editors more involved in the details of the module system, allow proposed changes to be made and evaluated on-the-fly by changing the portable implementation, and the resulting public-domain code

would greatly increase acceptance of R6RS by other implementors. It still remains to be seen if this portable implementation becomes a reality, as it represents quite a bit of work.

Dybvig noted that there are few differences between the module system proposed by Flatt and the one in Chez Scheme. This prompted an effort by Dybvig and Flatt to design a new module system that combines both systems. There has been a very active discussion since then.

We have made arrangements to have a whole-day meeting (September 18) in Snowbird to discuss these issues face to face. All editors will be there, except for Kelsey. We expect the module system to be the main topic of discussion and to make significant progress. We will also start discussing other issues on our list.